Implementing a Quantum Resistant Cryptosystem for MyFinance Inc.

Stanley Shaw

April 2025

Abstract

This report details a secure cryptosystem for *MyFinance Inc.*, an investment-portfolio platform. Confidentiality, integrity, and authenticity are enforced with AES-256-GCM, HKDF-derived keys, and Argon2 password hashing, while MLKEM-1024 provides post-quantum key exchange and rotation. Built on Django, the system combines role-based access control, HTTPS, audit logging, and end-to-end-encrypted user messaging. Unit and integration tests confirm the correctness of encryption, transaction security, and permission handling. Diagrams highlight that the solution is scalable, GDPR-compliant, and fit for modern financial workloads. Future work includes adding multifactor authentication and improving key lifecycle automation to enhance long-term resilience.

Contents

1 Introduction

MyFinance Inc.'s cryptosystem is designed to secure sensitive financial data and transactions by ensuring confidentiality, integrity, and authenticity. It employs a quantum-resistant lattice-based key encapsulation mechanism (MLKEM1024) to generate a shared secret key, which is then processed using HKDF to derive a robust symmetric key for AES-GCM encryption. The system further protects user credentials with a custom Argon2 hasher and guarantees data integrity through the built-in authentication tags of AES-GCM. Additional features such as secure HTTPS communication, role-based access control, key rotation, and audit logging together constitute a robust security framework for managing investment portfolios.

2 Design Justification

The design of the cryptosystem is built on industry best practices to address current cybersecurity threats and emerging quantum risks. It protects sensitive financial data from sophisticated attacks and unauthorised access by incorporating advanced, quantum-resistant algorithms, ensuring long-term confidentiality, robust security, and compliance with regulatory frameworks such as GDPR and the Data Protection Act (DPA) [Anderson, 2020, Mosca, 2018, European Union, 2018].

2.1 Functional Requirements

Functional requirements define the specific operations and capabilities the system must support to meet user needs. They ensure clear outlining of required functionalities across user roles.

User Role	Functional Requirements
Client	• **Register and Log In:** Clients are able to register and authenticate securely using robust authentication mechanisms [OWASP, 2025c, Django, 2023]. • **Access Account Details:** Clients view their encrypted account details and transaction history securely [PyCA, 2023]. • **Initiate Investment Transactions:** Clients initiate investment transactions, with all transaction data encrypted for security [NIST, 2001, Dworkin, 2007]. • **Modify Personal Information:** Clients update personal details (e.g., contact information) with encryption ensuring data confidentiality [Rescorla, 2018].
Financial Advisor	• **Secure Login:** Financial advisors log in using secure authentication mechanisms [OWASP, 2025c, Django, 2023]. • **View and Analyse Client Portfolios:** Advisors view and analyse client portfolios, with all portfolio data securely encrypted [NIST, 2001, PyCA, 2023]. • **Perform Client Transactions:** Advisors execute investment transactions on behalf of clients, ensuring that the data is encrypted during transmission [Dworkin, 2007]. • **Send Encrypted Communication:** Advisors send encrypted communications, such as investment recommendations, to clients [Rescorla, 2018].
System Admin	• **Manage User Accounts:** Administrators create and manage user accounts for clients and financial advisors [Django, 2023]. • **Monitor and Audit Logs:** Administrators monitor and audit system logs for any suspicious activities [Scarfone and Souppaya, 2006]. • **Oversee Key Management:** Administrators control the key management system to ensure secure generation and storage of cryptographic keys [Barker, 2020, NIST, 2024].

Table 1: Functional Requirements for the Cryptosystem

2.2 Non-Functional Requirements

Non-functional requirements define the quality attributes and operational constraints that the cryptosystem must meet to ensure performance, security, and reliability.

Non-Functional Requirement	**Description**
Encryption Methods	The system employs robust encryption techniques to secure client data and transaction details. It combines quantum-resistant and classical algorithms to ensure confidentiality against evolving threats [NIST, 2001, Dworkin, 2007, NIST, 2024].
Secure Communication	Secure communication channels are used to protect all data exchanged between MyFinance Inc. and its clients. Encryption is applied to data in transit, ensuring both confidentiality and integrity during transmission [Rescorla, 2018, OWASP, 2025b].
Key Management System	A secure key management system is in place for generating, distributing, and storing cryptographic keys. The design supports key rotation and advanced key protection measures, with further enhancements planned for future iterations [Barker, 2020, NIST, 2024].
User Authentication	A strong authentication mechanism is implemented to verify user identities. This includes secure password hashing and session management, ensuring that only authorised users can access sensitive functionalities [Biryukov et al., 2016, OWASP, 2025c].
Data Integrity	Measures are established to ensure the integrity of transaction data and prevent unauthorised modifications. Audit logging and encrypted transaction records help maintain the accuracy and reliability of financial data [NIST, 2015, OWASP, 2025a].

Table 2: Expanded Non-Functional Requirements for the Cryptosystem

2.3 Encryption and Hashing Algorithms

In order to align with both the functional and non-functional requirements, a range of encryption and hashing techniques that adhere closely to industry standards and best practices have been utilised.

Algorithm	**Justification**	**Suitability for Investment Platform**
AES-256	Symmetric encryption with a 256-bit key, providing extensive key space and resistance to brute-force attacks [NIST, 2001].	Essential for protecting sensitive financial information from unauthorised access [Dworkin, 2007].
Galois/ Counter Mode (GCM)	Authenticated encryption combining confidentiality with integrity assurance via authentication tags [Dworkin, 2007].	Ensures financial data remains unaltered and confidential during transmission [Rescorla, 2018].
HKDF	Derives secure, high-entropy cryptographic keys through extraction and expansion processes [Krawczyk and Eronen, 2010].	Guarantees secure, robust keys crucial for managing sensitive client data [PyCA, 2023].
SHA-256	Produces 256-bit hashes that are resistant to collisions, ensuring data integrity [NIST, 2015].	Critical in financial applications where data integrity is paramount to prevent fraud [Scarfone and Souppaya, 2006].
Argon2	Password hashing with adjustable complexity parameters to protect against brute-force attacks [Biryukov et al., 2016].	Secures client authentication credentials effectively, preventing unauthorised access [OWASP, 2025c].

Table 3: Encryption and Hashing Algorithms Justification

2.4 Key Management and Post-Quantum Cryptography

To support both security objectives and operational needs, a range of key management methods – including a quantum-resistant cryptographic approach – have been selected. These methods conform to current cryptographic standards and future-facing security models.

Component	**Detailed Description and Justification**	**Suitability for Investment Platform**
MLKEM-1024	MLKEM-1024 is a lattice-based post-quantum key encapsulation mechanism that relies on the hardness of the Module Learning With Errors (MLWE) problem. It provides IND-CCA2 security, offering resistance to both classical and quantum attacks. As part of the Kyber family, it is efficient in terms of speed and memory, making it practical for real-world use [NIST, 2024].	Provides robust encryption that protects against future quantum decryption.
Key Pair Generation	A pseudorandom seed generates a public matrix, while small noise vectors are sampled to create the secret and error terms. The public key results from a noisy matrix multiplication, while the secret key includes the noise vectors and hashes needed for verification. This ensures strong one-wayness under the MLWE assumption [NIST, 2024].	Provides quantum-secure cryptographic identities for users and services.
Key Encapsulation	The sender encrypts a randomly generated message using the recipient's public key to form a ciphertext. A shared secret is then derived by hashing both the message and the ciphertext. This ensures that the session key cannot be predicted or reconstructed without the correct private key [NIST, 2024].	Secures session key exchange during authentication or transactions.
Key Decapsulation	The recipient uses their private key to decrypt the ciphertext and recover the original message. A re-encapsulation check ensures message integrity and prevents chosen-ciphertext attacks. If the check fails, fallback handling protects against side-channel leakage [NIST, 2024].	Ensures only legitimate recipients can access confidential session keys.
Key Rotation	Key material is periodically refreshed by generating new key pairs. This reduces exposure in case of compromise and supports cryptographic agility. It also aligns with best practices for forward secrecy and regulatory compliance [Barker, 2020].	Helps maintain long-term confidentiality and risk mitigation.

Table 4: MLKEM-1024 and Key Management for Investment Platform Security

2.5 User Authentication and Access Control

To meet both the functional and non-functional requirements, a range of user authentication mechanisms and access controls are implemented in accordance with industry best practices, ensuring that only authorised users can access sensitive financial data.

Feature	Detailed Explanation	Suitability for Investment Platform
Robust Authentication	Utilises Argon2 hashing for secure storage of user credentials [Biryukov et al., 2016, OWASP, 2025c].	Protects against unauthorised financial data access by securing user credentials [Django, 2023].
Role-Based Permissions	Defines clear user roles (client, advisor, admin) for appropriate access control [Sandhu et al., 1996].	Prevents data misuse by limiting access according to role-specific financial functions [Django, 2023].
Granular Access Control	Ensures permissions precisely reflect authorised activities [Sandhu et al., 1996].	Reduces risk of internal breaches by strictly controlling access to sensitive investment data [Django, 2023].
Secure Session Management	Implements timeouts and session ID regeneration [Django, 2023].	Protects investment sessions from session hijacking attacks and unauthorised user impersonation [Rescorla, 2018].
Audit Logging	Comprehensive logging of access and security events [Scarfone and Souppaya, 2006].	Facilitates compliance audits and incident investigations in financial services [OWASP, 2025a].

Table 5: User Authentication and Access Control Mechanisms

2.6 Data Integrity and Secure Communication

Robust measures are implemented to ensure data integrity and secure communication channels, safeguarding financial data against unauthorised modifications and potential cyber threats.

Feature	Justification and Detailed Explanation	Suitability for Investment Platform
HMAC	Detects unauthorised data modifications using SHA-256 signatures [NIST, 2008].	Critical for ensuring integrity of transaction records and financial data [NIST, 2015].
Collision Resistance	Ensures small changes produce distinct hash results, preventing undetected tampering [NIST, 2015].	Protects financial records from subtle fraud attempts [Scarfone and Souppaya, 2006].
HTTPS	HTTPS encrypts all traffic between client and server, offering protection against data interception, tampering, and eavesdropping [Rescorla, 2018].	Essential for secure online financial transactions and client interactions [OWASP, 2025b].
TLS	Implements strong encryption and validation for HTTPS [Rescorla, 2018].	Provides additional security layer for sensitive financial communications [OWASP, 2025b].
End-to-End Encryption	Layered encryption for data at rest and in transit [PyCA, 2023].	Ensures comprehensive data protection essential for client trust in financial services [Dworkin, 2007].

Table 6: Data Integrity and Secure Communication

2.7 System Diagrams

This section presents system diagrams illustrating the core cryptographic processes—encryption, decryption, password hashing—and system structure. These diagrams demonstrate how the implementation aligns with industry standards and secure design practices.

Class Diagram

The following diagram illustrates the classes within my financial project. It shows how each class interacts with each other through functions in order to create the final investment platform.

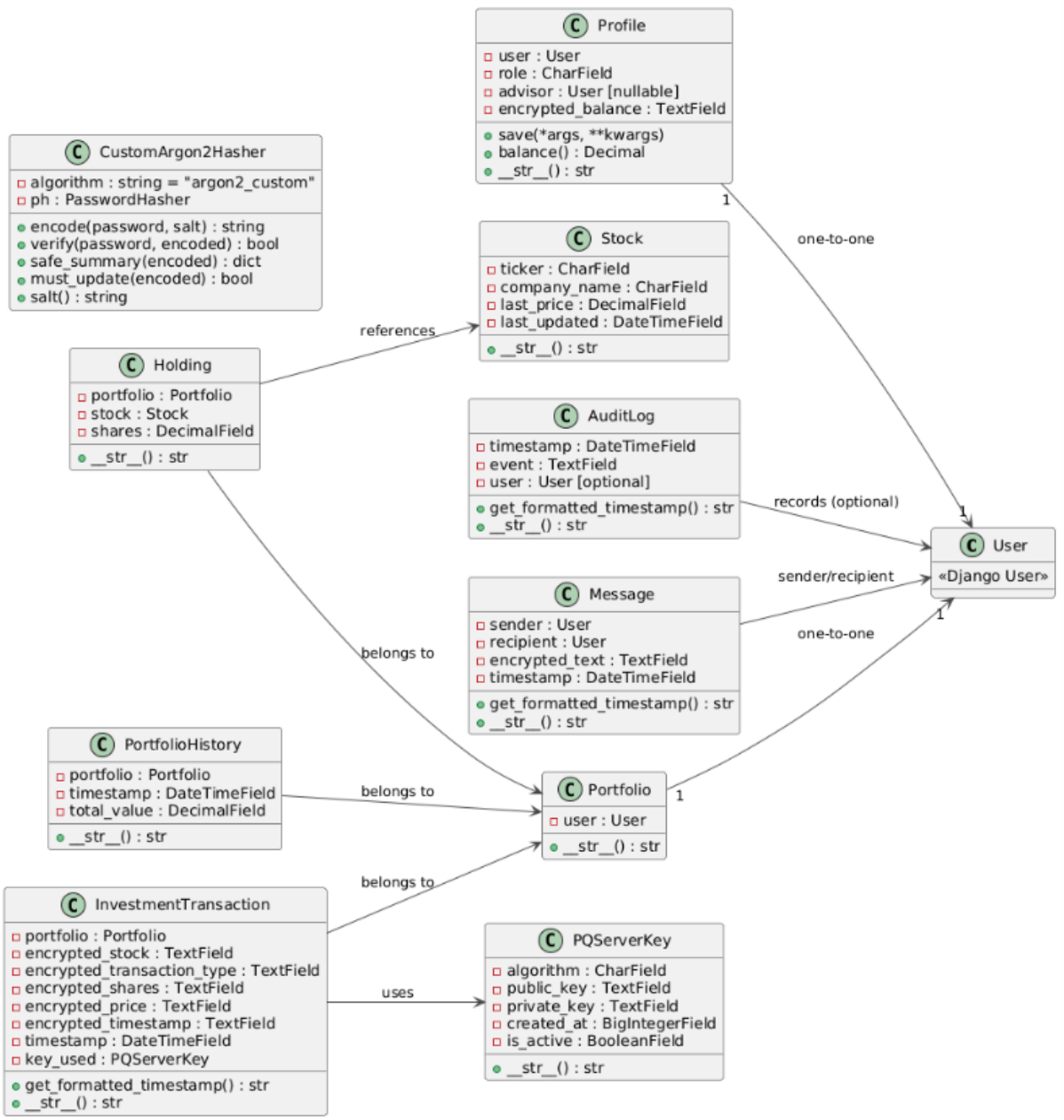

Figure 1: Class diagram highlighting key functions and classes

Encryption Process Activity Diagram

The following activity diagram illustrates the steps involved in encrypting a message (e.g., via the `encrypt_message` function). It shows how the system retrieves or generates a key, derives a symmetric key using HKDF, and uses AES-GCM for encryption.

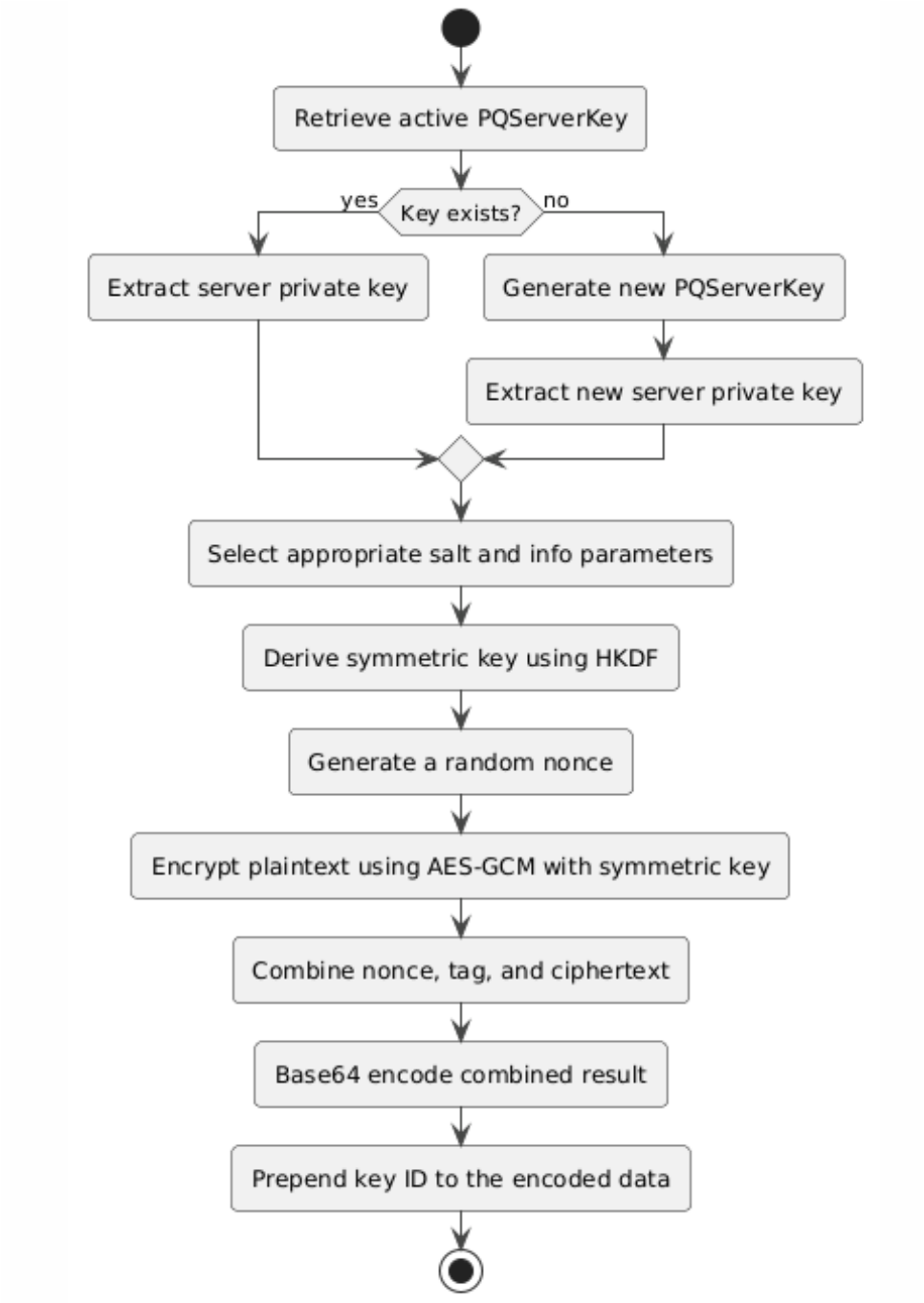

Figure 2: Encryption Activity Diagram

Decryption Process Activity Diagram

This activity diagram details the decryption process. The process involves extracting the key identifier, retrieving the corresponding key, deriving the symmetric key, and finally decrypting the message.

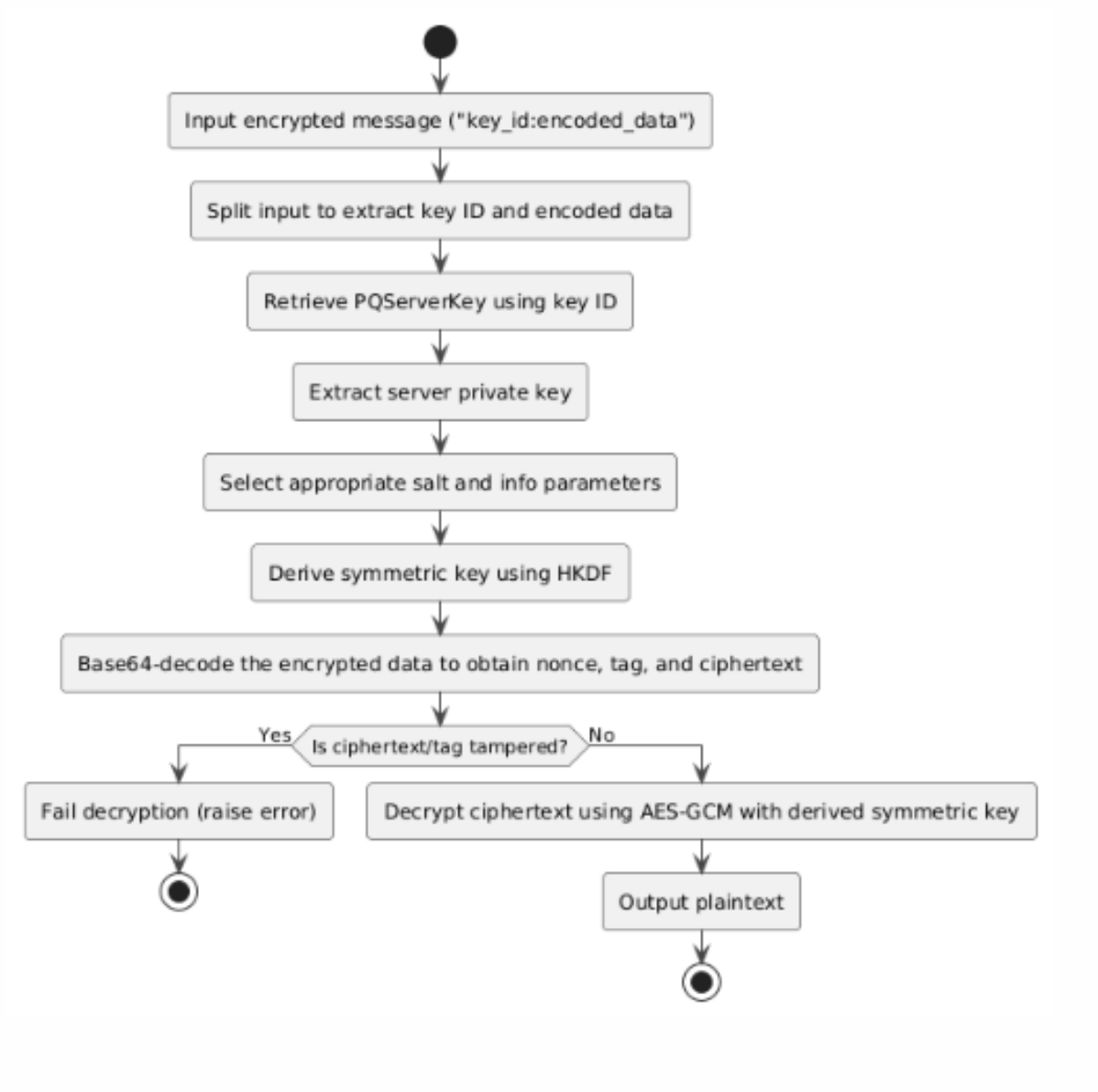

Figure 3: Decryption Activity Diagram

Password Hashing Process Activity Diagram

The following activity diagram explains the password hashing process using the custom Argon2 implementation. It shows how the plain password is processed, salted, hashed, and then stored for secure user authentication.

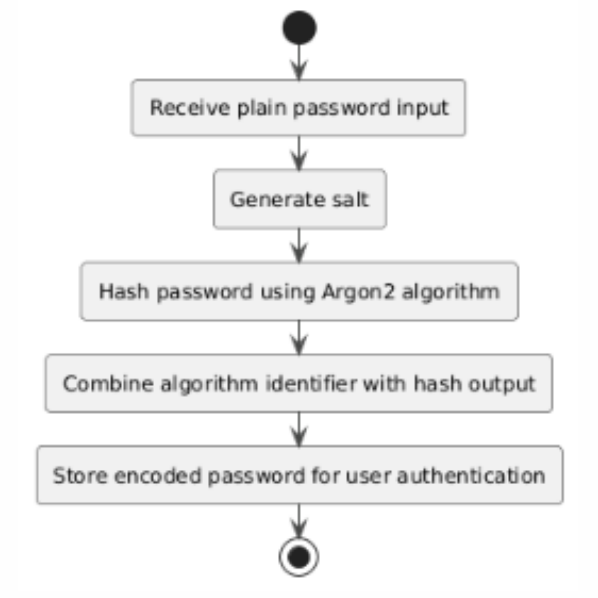

Figure 4: Password Hashing Activity Diagram

Key Rotation Process Activity Diagram

The following activity diagram explains the key rotation process using the MLKEM1024 algorithm. It shows how the current key is deactivated, a new key pair is generated, encoded, stored, and the rotation is logged for auditing.

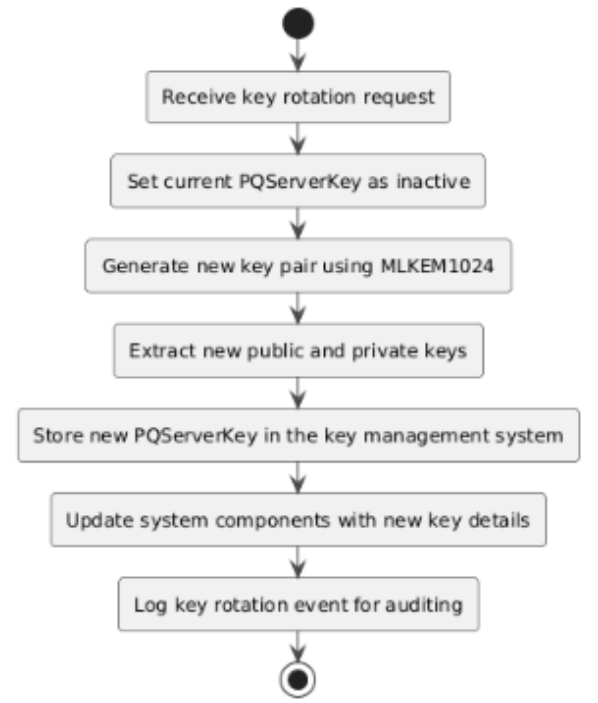

Figure 5: Key Rotation Activity Diagram

Component Diagram

This following component diagram illustrates a three-layer architecture—UI, business logic, and data—highlighting how user requests flow from the interface (UI/Browser) through core application processes (encryption, hashing, data handling) to storage (Cloud/SQL Server), ensuring secure data transmission and robust permission controls.

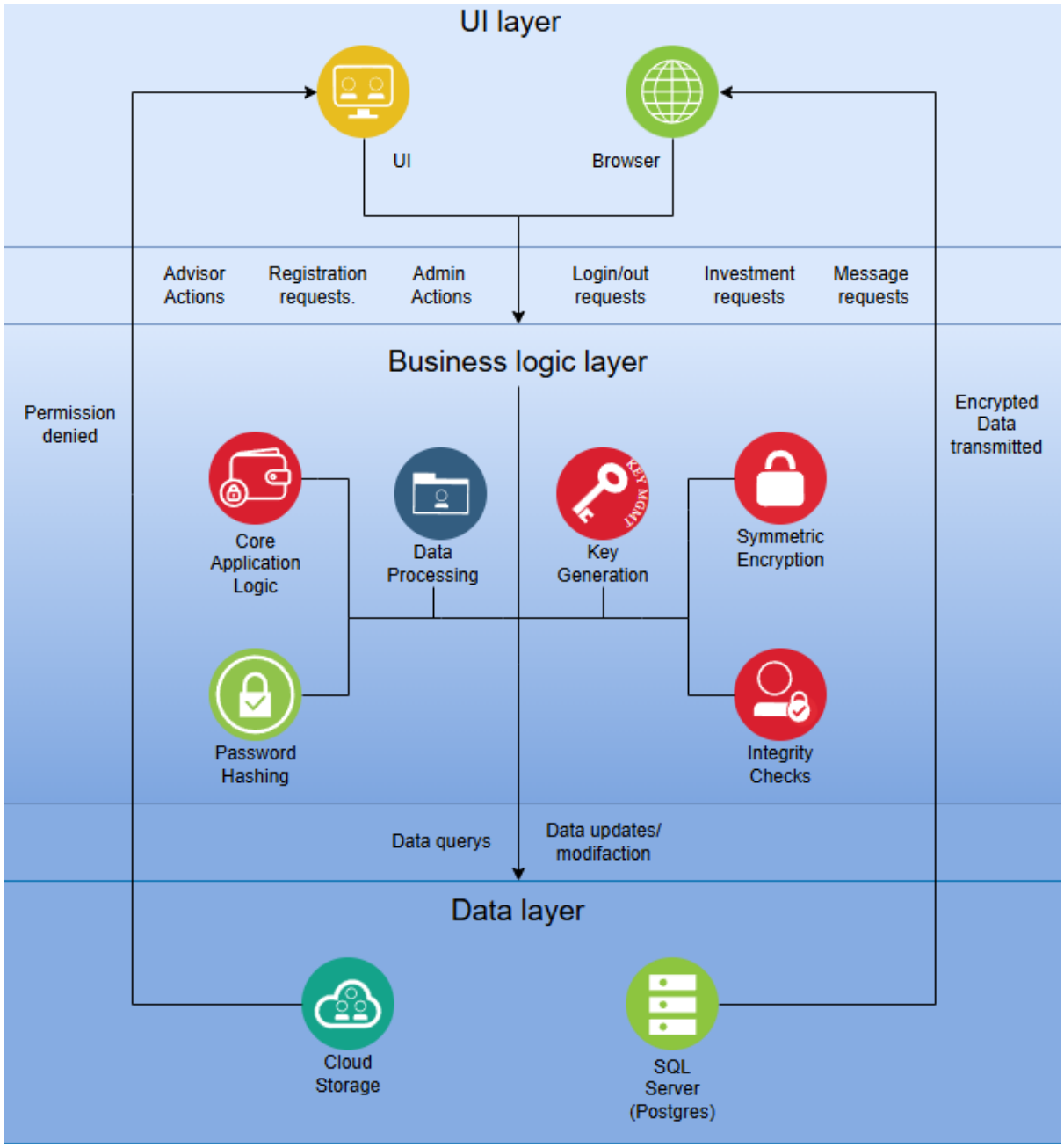

Figure 6: Three-Tier Architecture with Security Controls

3 Implementation Details

This section details how the system secures financial data using Django and advanced cryptographic techniques. The implementation relies on key libraries such as `cryptography` [PyCA, 2023] and `quantcrypt` [Aabmets, 2024] (for encryption and key derivation), and `yfinance` [Aroussi, 2025] (for live stock data). The following subsections describe each core component, with corresponding figures that highlight the code.

3.1 Key Management

Key management is critical for long-term security. The system uses MLKEM-1024 for quantum-resistant key encapsulation [NIST, 2024]. A new key pair (public and private) is generated, stored securely in the database, and rotated periodically. **Figure 7** illustrates the key management functions implemented to support post-quantum security, while **Figure 8** provides a working implementation of key generation and storage.

```python
ALGORITHM = "MLKEM1024"
_server_keys_cache = None # Caching server keys to avoid redundant queries
def get_server_keys():
    global _server_keys_cache #Retrieve or generate server keys (cached)
    if _server_keys_cache:
        return _server_keys_cache
    from .models import PQServerKey
    key = PQServerKey.objects.filter(is_active=True).first()
    if not key:
        key = generate_new_key()
    _server_keys_cache = (
        key.algorithm,
        base64.b64decode(key.public_key),
        base64.b64decode(key.private_key)
    )
    return _server_keys_cache

def generate_new_key():
    from .models import PQServerKey #Generates and saves a new PQServerKey
    kem_algorithm = MLKEM_1024()
    public_key, private_key = kem_algorithm.keygen()
    return PQServerKey.objects.create(
        algorithm=ALGORITHM,
        public_key=base64.b64encode(public_key).decode('utf-8'),
        private_key=base64.b64encode(private_key).decode('utf-8'),
        created_at=int(time.time()),
        is_active=True,
    )
```

Figure 7: Post Quantum Key Management Functions

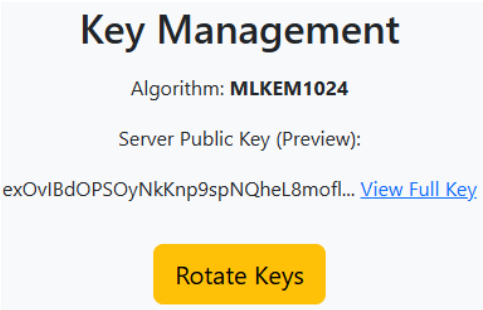

Figure 8: Post Quantum Key Management Working Example

3.2 AES Encryption and Decryption

Symmetric encryption is implemented in `crypto_utils.py` using AES-GCM [Dworkin, 2007, NIST, 2001], which ensures both confidentiality and integrity.

Deriving the Symmetric Key

Symmetric keys are derived from the MLKEM-1024 private key using HKDF with SHA-256, a fixed salt, and a context-specific information string [Krawczyk and Eronen, 2010]. **Figure 9** illustrates this key derivation process.

```python
def derive_symmetric_key(salt: bytes, info: bytes, server_priv=None) -> bytes:
    #Derives a symmetric key using HKDF with the active server key.
    if server_priv is None:
        _, _, server_priv = get_server_keys()

    return HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        info=info,
        backend=default_backend()
    ).derive(server_priv)
```

Figure 9: Derive Symmetric Key Function

Encryption and Decryption Processes

Encryption converts input text (UTF-8 encoded) into ciphertext using AES-GCM [Dworkin, 2007]. A random nonce is generated to prevent replay attacks, and the nonce, authentication tag, and ciphertext are concatenated and Base64-encoded. **Figure 10** demonstrates the encryption process, while **Figure 11** shows how the system reverses this operation during decryption.

```python
def encrypt_data(value: str, salt: bytes, info: bytes) -> str:
    """Encrypts data using AES-GCM."""
    symmetric_key = derive_symmetric_key(salt, info)
    plaintext = value.encode('utf-8')
    nonce = os.urandom(12)
    cipher = Cipher(algorithms.AES(symmetric_key), modes.GCM(nonce), backend=default_backend())
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(plaintext) + encryptor.finalize()
    combined = nonce + encryptor.tag + ciphertext
    return base64.b64encode(combined).decode('utf-8')
```

Figure 10: Encrypt Data Function

```python
def decrypt_data(encrypted_value: str, salt: bytes, info: bytes, server_priv=None) -> str:
    """Decrypts AES-GCM encrypted data."""
    symmetric_key = derive_symmetric_key(salt, info, server_priv)
    combined = base64.b64decode(encrypted_value)
    nonce = combined[:12]
    tag = combined[12:28]
    ciphertext = combined[28:]
    cipher = Cipher(algorithms.AES(symmetric_key), modes.GCM(nonce, tag), backend=default_backend())
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()
    return plaintext.decode('utf-8')
```

Figure 11: Decrypt Data Function

Balance Encryption and Decryption

To protect sensitive balance data, the system encrypts the balance by converting it to a string and prefixing the ciphertext with the active key's ID. This enables retrieval of the correct key during decryption [PyCA, 2023]. **Figure 12** shows the balance encryption function, and **Figure 13** depicts the corresponding decryption. Additionally, **Figure 14** details how the encrypted balance is stored, while **Figure 15** and **Figure 16** illustrate how the *Profile* model automatically decrypts and displays the balance.

```python
def encrypt_balance(balance: Decimal) -> str:
    from .models import PQServerKey
    key_obj = PQServerKey.objects.filter(is_active=True).first() or generate_new_key()
    key_id = key_obj.id
    encrypted_data = encrypt_data(str(balance), b'balance_salt', b'balance encryption')
    return f"{key_id}:{encrypted_data}"
```

Figure 12: Encrypt Balance Function

```python
def decrypt_balance(encrypted_balance: str) -> Decimal:
    from .models import PQServerKey
    key_id_str, encoded_data = encrypted_balance.split(":", 1)
    key_id = int(key_id_str)
    try:
        key_obj = PQServerKey.objects.get(id=key_id)
    except PQServerKey.DoesNotExist:
        raise Exception("Encryption key not found.")
    server_priv = base64.b64decode(key_obj.private_key)
    decrypted_value = decrypt_data(encoded_data, b'balance_salt', b'balance encryption', server_priv)
    return Decimal(decrypted_value)
```

Figure 13: Decrypt Balance Function

id [PK] bigint	role character varying (20)	encrypted_balance text
1	admin	0tc8HCdAA06oGdhhG/OhxrEt/emg16KW52q4Ki5CNVZL
2	advisor	cnT/XlPqWTw0KPN9lH7M0Wk9FNivTNiUXTKBc/6iHxtZ
7	client	lor6N5vDKgm4HA8ESxxs6TK2Xzm74+KymNv1QkbV3cEW

Figure 14: Balance Encryption Implementation

```python
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    role = models.CharField(max_length=20, choices=ROLE_CHOICES, default='client')
    advisor = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True, related_name='clients_assigned')
    encrypted_balance = models.TextField(default='')

    def save(self, *args, **kwargs):

        if not self.encrypted_balance:
            self.encrypted_balance = encrypt_balance(Decimal('10000'))
        super().save(*args, **kwargs)

    @property
    def balance(self):
        try:
            return decrypt_balance(self.encrypted_balance)
        except Exception:
            return Decimal('0')

    @balance.setter
    def balance(self, value):
        self.encrypted_balance = encrypt_balance(Decimal(value))

    def __str__(self):
        return self.user.username
```

Figure 15: Balance Decryption Implementation in Profile Model

Portfolio Distribution

Total Portfolio Value: £10000.00

Asset	Value (£)	Percentage (%)
Cash	£4910.68	49.11%
PG	£500.07	5.00%
TSLA	£3979.36	39.79%
VZ	£87.98	0.88%
BRK-B	£521.91	5.22%

Figure 16: Balance Decryption Working Example

Sensitive Field Encryption and Decryption

General-purpose functions encrypt and decrypt other sensitive fields in the database [PyCA, 2023]. **Figure 17** shows these functions, and **Figure 18** provides an example of encrypted values. While functional decryption can be seen in the admin dashboard within **Figures 19**, **20**, and **21**.

```python
# FIELD ENCRYPTION / DECRYPTION
def encrypt_field(value: str, salt: bytes, info: bytes) -> str:
    return encrypt_data(value, salt, info)

def decrypt_field(encrypted_value: str, salt: bytes, info: bytes, server_priv=None) -> str:
    return decrypt_data(encrypted_value, salt, info, server_priv)
```

Figure 17: Decrypt/Encrypt Field Functions

encrypted_stock text	encrypted_transaction_type text	encrypted_shares text
7PGTxKGS/PyApZaThIYX4gE9i9vm3PHdxR4Sze...	zq+uL2wHg8kVvPYkCniZAnGuydJwrXgzCGjKEB+8U...	QfYkF3yc81GLv3w+FngZiiqGXdDYfRMETGXEQWI=
nIko4S2HKJ5GKO+RXThISDn+EFUIkf0eB6VVaF6E	/7Iv4ytANS7O1sAtNJ5HqD/Px8Rvr+DP63b8rvMLbw==	mLEE/h+xyJxgAUn8DfKWQ0Y/BXPjcd0KfKRp99c=

Figure 18: Encrypted Fields Implementation

```python
transactions = InvestmentTransaction.objects.all().order_by('-timestamp')
for tx in transactions[:5]:
    try:
        # Use the correct key:
        if tx.key_used:
            local_priv = base64.b64decode(tx.key_used.private_key)
        else:
            # fallback to the currently active key
            local_priv = active_server_priv

        # Now decrypt each field with that private key
        decrypted_stock = decrypt_field(tx.encrypted_stock, b'investment_stock',
                                        b'investment stock encryption', local_priv)
        decrypted_type = decrypt_field(tx.encrypted_transaction_type, b'investment_type',
                                       b'investment type encryption', local_priv)
        shares_str = decrypt_field(tx.encrypted_shares, b'investment_shares',
                                   b'investment shares encryption', local_priv)
        price_str = decrypt_field(tx.encrypted_price, b'investment_price',
                                  b'investment price encryption', local_priv)
```

Figure 19: Decrypt Transactions Function

```
<!-- Recent Transactions Section -->
<div class="mt-4">
  <h2 class="text-center mb-3">Recent Transactions</h2>
  <div class="table-responsive">
    <table class="table table-bordered table-striped text-center">
      <thead class="table-dark">
        <tr>
          <th>ID</th><th>Stock</th><th>Type</th><th>Shares</th><th>Price (£)</th><th>Timestamp</th>
        </tr>
      </thead>
      <tbody>
        {% for tx in recent_transactions %}
          <tr>
            <td>{{ tx.id }}</td>
            <td>{{ tx.stock }}</td><td>{{ tx.transaction_type }}</td><td>{{ tx.shares|floatformat:2 }}</td>
            <td>£{{ tx.price|floatformat:2 }}</td><td>{{ tx.timestamp }}</td>
          </tr>
        {% empty %}
          <tr>
            <td colspan="6">No recent transactions found.</td>
          </tr>
        {% endfor %}
      </tbody>
    </table>
    </div>
  </div>
```

Figure 20: Decrypted Transactions HTML Template

Recent Transactions

ID	Stock	Type	Shares	Price (£)	Timestamp
8	NFLX	SELL	2.00	£960.29	2025-03-23 15:19:56
7	XOM	BUY	3.00	£115.50	2025-03-23 15:07:09
6	NFLX	BUY	4.00	£960.29	2025-03-23 15:07:01
5	GOOGL	BUY	3.00	£163.99	2025-03-23 15:06:57
4	JPM	BUY	3.00	£241.63	2025-03-23 15:06:53

Figure 21: Decrypted Transactions Working Implementation

Message Encryption and Decryption

For internal communications, messages are encrypted using a symmetric key derived with HKDF (using dedicated parameters for messages) [Krawczyk and Eronen, 2010]. The active key is retrieved (or generated) and its ID is prepended to the ciphertext. During decryption, the key ID is used to fetch the correct key [PyCA, 2023]. **Figures 22** and **23** show the encryption and decryption functions, while **Figure 24** provides an example of an encrypted message. **Figure 25** displays the HTML for chat integration, and **Figure 26** demonstrates functional chat integration.

```python
def encrypt_message(message: str) -> str:
    #Encrypts a message with key ID tracking for key rotation support.
    from .models import PQServerKey
    key_obj = PQServerKey.objects.filter(is_active=True).first() or generate_new_key()
    key_id = key_obj.id
    encrypted_data = encrypt_data(message, b'message_salt', b'message encryption')
    return f"{key_id}:{encrypted_data}"
```

Figure 22: Encrypt Message Function

```python
def decrypt_message(encrypted_message: str) -> str:
    #Decrypts a message using the key ID for compatibility with rotated keys.
    key_id_str, encoded_data = encrypted_message.split(":", 1)
    key_id = int(key_id_str)
    # Retrieve the correct key based on the stored key ID
    from .models import PQServerKey
    try:
        key_obj = PQServerKey.objects.get(id=key_id)
    except PQServerKey.DoesNotExist:
        raise Exception("Encryption key not found.")
    server_priv = base64.b64decode(key_obj.private_key)
    return decrypt_data(encoded_data, b'message_salt', b'message encryption', server_priv)
```

Figure 23: Decrypt Message Function

encrypted_text text	timestamp timestamp with time zone	recipient_id integer	sender_id integer
LSfjMcOtgKA7cPXumkP7WcudPyKZNHB61+feyuN66RY84Q==	2025-03-23 15:04:38.283653+00	2	1
z4S3Iq2H9CKdjzz5HJxntWOGaAMr+Nj9zT7BKG3174h0	2025-03-23 16:24:43.362214+00	1	2
5:Hn1FXx5NHCwIc2cR1QgZf1mzeE3WwKSrJomchiVVZKZg	2025-03-23 16:28:54.272776+00	2	1

Figure 24: Example of an Encrypted Message

```
{% extends "base.html" %}
{% block content %}
<div class="container my-5">
  <h2 class="mb-4">Your Conversations</h2>
  <ul class="list-group">
    {% for partner in partners %}
      <li class="list-group-item">
        <a href="{% url 'chat' partner.username %}">{{ partner.username }}</a>
      </li>
    {% empty %}
      <li class="list-group-item">No conversations yet.</li>
    {% endfor %}
  </ul>
</div>
{% endblock %}
```

Figure 25: HTML for Chat Implementation

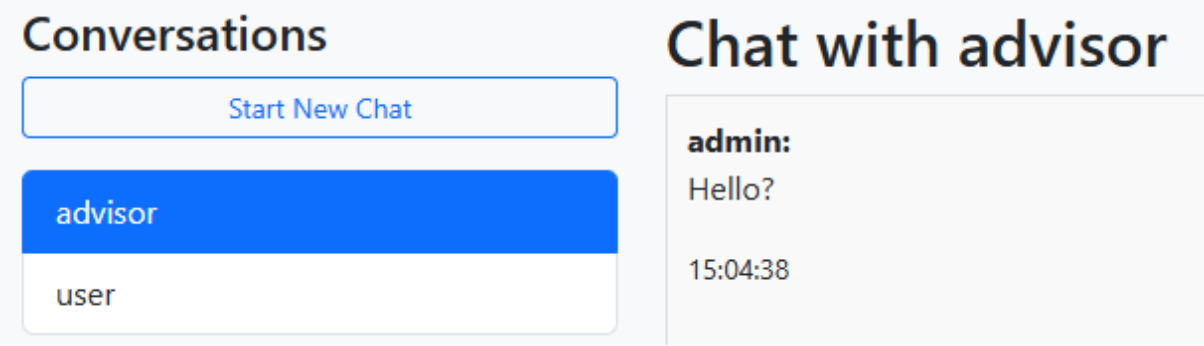

Figure 26: Functional Chat Integration

3.3 Password Hashing

A custom Argon2-based hasher is used to secure passwords [Biryukov et al., 2016, OWASP, 2025c]. It enables precise control over time and memory parameters to resist brute-force attempts. New passwords are hashed using this method. **Figure 27** displays the custom Argon2 hashing function, while **Figure 28** demonstrates a working example.

```python
class CustomArgon2Hasher(BasePasswordHasher):
    algorithm = 'argon2_custom'

    # Custom Argon2 Configuration
    ph = PasswordHasher(
        time_cost=4,        # Increases computational cost (default: 2)
        memory_cost=102400, # Memory usage in KB (default: 512)
        parallelism=2,      # Parallelism (default: 2)
        hash_len=32,        # Length of the output hash
        salt_len=16         # Length of the generated salt
    )

    def encode(self, password, salt):
        assert password is not None
        assert salt and '$' not in salt  # Ensure no accidental `$` symbols
        hashed_password = self.ph.hash(password)
        return f"{self.algorithm}${hashed_password}"

    def verify(self, password, encoded):
        algorithm, hash_str = encoded.split('$', 1)
        try:
            return self.ph.verify(hash_str, password)
        except Exception:
            return False

    def safe_summary(self, encoded):
        algorithm, hash_str = encoded.split('$', 1)
        return {
            'algorithm': algorithm,
            'hash': hash_str[:6] + '...' + hash_str[-6:]  # Partial hash for security
        }

    def must_update(self, encoded):
        algorithm, hash_str = encoded.split('$', 1)
        try:
            return self.ph.check_needs_rehash(hash_str)
        except Exception:
            return True  # Assume update is needed if check fails

    def salt(self):
        return base64.b64encode(self.ph.hash("generate_salt").encode('utf-8'))[:16].decode('utf-8')
```

Figure 27: Custom Argon2 Hashing Function

password character varying (128)
argon2_custom$$argon2id$v=19$m=102400,t=4,p=2$nCQpJsIpcRvIxMK/BWc8KA$xigZdWIBOPPmBqVroU5jifPJI9mPLuAt4vZXvqH4/KE
argon2_custom$$argon2id$v=19$m=102400,t=4,p=2$6PiiKHxiNr3HnO2yYLO6ow$oBMfRY7+kdd/kQBhFXaj994ldK528sKK9Nc6InREPvY
argon2_custom$$argon2id$v=19$m=102400,t=4,p=2$TYsLwF9t5gmWO3fzmcbQnQ$OQ3IEYOyr/8/hQBOHR/iABzv57zhGIPJs12QqWQQQCg

Figure 28: Custom Argon2 Password Hashing Working Example

3.4 Secure Communication and Audit Logging

Secure client-server communication is enforced via HTTPS/TLS [Rescorla, 2018]. The web server uses SSL certificates (configured as shown in **Figure 29**) to encrypt traffic. Comprehensive audit logging is achieved via the `AuditLog` model, which records critical operations along with event details, timestamps, and responsible users [Scarfone and Souppaya, 2006, OWASP, 2025a]. These logs are displayed on the admin dashboard to facilitate monitoring and auditing, as illustrated in **Figures 30**, **31**, **32**, and **33**. Furthermore, rate limiting was implemented to prevent brute force and DDOS attacks through the *django-ratelimit* module [Django, 2023] as seen in **Figures 34 and 35**.

```
            'sslmode': env('DB_SSLMODE'),
            'sslrootcert': env('DB_SSLCERT'),
        },
    }
}
SECURE_HSTS_SECONDS = 31536000
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True
SECURE_SSL_REDIRECT = True
```

Figure 29: Web App SSL Configuration

```
# Audit log for admin actions
class AuditLog(models.Model):
    timestamp = models.DateTimeField(auto_now_add=True)
    event = models.TextField()
    user = models.ForeignKey(User, null=True, blank=True, on_delete=models.SET_NULL)
    def get_formatted_timestamp(self):
        return self.timestamp.strftime("%Y-%m-%d %H:%M:%S")
    def __str__(self):
        return f"{self.get_formatted_timestamp()} - {self.event}"
```

Figure 30: Logging Model Implementation

```
@login_required
def admin_dashboard_view(request):
    if request.user.profile.role != 'admin':
        messages.error(request, "Access denied.")
        return redirect('portfolio')
    users = User.objects.all().order_by('username')
    total_users = users.count()
    algorithm, server_pub, server_priv = get_server_keys()
    server_public_key = base64.b64encode(server_pub).decode('utf-8')
    transactions = InvestmentTransaction.objects.all().order_by('-timestamp')
    total_transactions = transactions.count()
    total_money = sum(tx.shares * tx.price for tx in transactions) if total_transactions > 0 else 0
    average_transaction = total_money / total_transactions if total_transactions > 0 else 0
    audit_logs = AuditLog.objects.all().order_by('-timestamp')[:10]
    context = {
        'users': users,
        'total_users': total_users,
        'server_algorithm': algorithm,
        'server_public_key': server_public_key,
        'analytics': {
            'total_transactions': total_transactions,
            'total_money_moved': total_money,
            'average_transaction': average_transaction,
        },
        'recent_transactions': transactions[:5],
        'audit_logs': audit_logs,
    }
    return render(request, 'admin_dashboard.html', context)
```

Figure 31: Admin View Logs Function

```
<!-- Audit Logs Section -->
<div class="mb-5">
  <h2>Audit Logs</h2>
  <div class="table-responsive">
    <table class="table table-bordered table-striped text-center">
      <thead class="table-dark">
        <tr>
          <th>Timestamp</th>
          <th>Event</th>
          <th>User</th>
        </tr>
      </thead>
      <tbody>
        {% for log in audit_logs %}
          <tr>
            <td>{{ log.get_formatted_timestamp }}</td>
            <td>{{ log.event }}</td>
            <td>{% if log.user %}{{ log.user.username }}{% else %}N/A{% endif %}</td>
          </tr>
        {% empty %}
          <tr>
            <td colspan="3">No audit logs found.</td>
          </tr>
        {% endfor %}
      </tbody>
    </table>
  </div>
</div>
```

Figure 32: HTML to Display Audit Logs

Audit Logs

Timestamp	Event	User
2025-03-23 16:08:16	Encryption keys rotated. Old keys archived, new key generated.	admin
2025-03-23 15:52:48	Admin admin created user user.	admin
2025-03-23 15:52:32	User user deleted by admin admin.	admin
2025-03-23 15:52:28	User bob deleted by admin admin.	admin
2025-03-23 15:52:10	Admin admin created user bob.	admin
2025-03-23 15:51:55	User bob deleted by admin admin.	admin
2025-03-23 15:49:46	User admin logged in.	admin
2025-03-23 15:49:39	User admin logged out.	admin

Figure 33: Working Audit Logs on Admin Dashboard

```python
# bank/middleware.py

from django.core.cache import cache
from django.http import HttpResponse
from django.utils.deprecation import MiddlewareMixin

def check_rate_limit(request, rate='100/h'):
    try:
        count_str, period = rate.split('/')
        count = int(count_str)
    except ValueError:
        # Fallback to a default if rate is misconfigured
        count = 100
        period = 'h'
    # Determine the period in seconds
    if period == 'h':
        seconds = 3600
    elif period == 'm':
        seconds = 60
    else:
        seconds = 1  # Default: 1 second
    ip = request.META.get('REMOTE_ADDR', 'unknown')
    cache_key = f"ratelimit:{ip}"

    current = cache.get(cache_key)
    if current is None:
        cache.set(cache_key, 1, timeout=seconds)
        return False
    else:
        if current >= count:
            return True  # Rate limit exceeded
        else:
            try:
                # Increase the counter
                cache.incr(cache_key)
            except ValueError:
                # In case the key expired between get and incr, set it again.
                cache.set(cache_key, 1, timeout=seconds)
            return False

class GlobalRateLimitMiddleware(MiddlewareMixin):
    def process_view(self, request, view_func, view_args, view_kwargs):
        # Check rate limit: set to 100 requests per minute per IP
        if check_rate_limit(request, rate='100/m'):
            return HttpResponse("Too many requests. Please try again later.", status=429)
        # Continue processing the view if limit not exceeded.
        return None
```

Figure 34: Rate Limiting Implementation

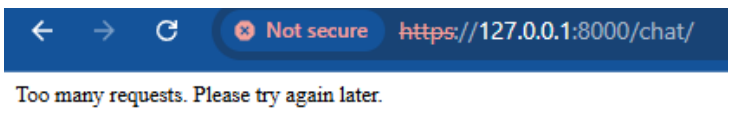

Figure 35: Rate Limiting Working Example

3.5 Integration with Django and Interface Creation

The integration of cryptographic functions with Django follows a natural user journey: authentication, data display, transactions, and role-specific dashboards [Django, 2023].

User Authentication Setup

User registration and login are secured using the custom Argon2 hasher [Biryukov et al., 2016]. Dedicated views and HTML templates handle these processes. **Figures 36**, **37**, **38**, **39**, **40**, and **41** provide examples of the login and registration implementation and their respective working examples.

```python
def login_view(request):
    if request.method == 'POST':
        username = request.POST.get("username").strip()
        password = request.POST.get("password").strip()
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            AuditLog.objects.create(
                event=f"User {username} logged in.",
                user=user
            )
            messages.success(request, f"Welcome {username}!")
            if user.profile.role == 'admin':
                return redirect('admin_dashboard')
            elif user.profile.role == 'advisor':
                return redirect('advisor')
            else:
                return redirect('portfolio')
        else:
            messages.error(request, "Invalid credentials!")
            return redirect('login')
    return render(request, 'login.html')
```

Figure 36: Login View Function

```python
def register_view(request):
    if request.method == 'POST':
        form = RegistrationForm(request.POST)
        if form.is_valid():
            user = form.save()
            user.profile.role = form.cleaned_data.get('role')
            if user.profile.role == 'client':
                user.profile.advisor = form.cleaned_data.get('advisor')
            user.profile.save()

            AuditLog.objects.create(
                event=f"User {user.username} registered with role {user.profile.role}.",
                user=user
            )
            messages.success(request, "Registration successful. Please log in.")
            return redirect('login')
        else:
            for field in form:
                for error in field.errors:
                    messages.error(request, f"{field.label}: {error}")
            for error in form.non_field_errors():
                messages.error(request, error)
            return redirect('register')
    else:
        form = RegistrationForm()
    return render(request, 'register.html', {'form': form})
```

Figure 37: Register View Function

```html
{% extends "base.html" %}
{% block content %}
<div class="card mx-auto" style="max-width: 500px;">
  <div class="card-body">
    <h3 class="card-title mb-3 text-center">Login</h3>
    <form method="POST">
      {% csrf_token %}
      <div class="mb-3">
        <label for="username" class="form-label">Username</label>
        <input class="form-control" type="text" name="username" id="username" required>
      </div>
      <div class="mb-3">
        <label for="password" class="form-label">Password</label>
        <input class="form-control" type="password" name="password" id="password" required>
      </div>
      <button class="btn btn-primary w-100" type="submit">Login</button>
    </form>
    <div class="text-center mt-3">
      <a href="{% url 'password_reset' %}">Forgot Password?</a>
    </div>
  </div>
</div>
{% endblock %}
```

Figure 38: Login View HTML Template

```
{% extends "base.html" %}
{% load widget_tweaks %}
{% block content %}
<div class="container d-flex justify-content-center align-items-center" style="min-height: 80vh;">
  <div class="card" style="width: 100%; max-width: 500px;">
    <div class="card-body">
      <h3 class="card-title mb-3 text-center">Register</h3>
      <form method="POST">
        {% csrf_token %}
        <div class="mb-3">
          {{ form.username.label_tag }}
          {{ form.username|add_class:"form-control" }}
        </div>
        <div class="mb-3">
          {{ form.email.label_tag }}
          {{ form.email|add_class:"form-control" }}
        </div>
        <div class="mb-3">
          {{ form.password1.label_tag }}
          {{ form.password1|add_class:"form-control" }}
        </div>
        <div class="mb-3">
          {{ form.password2.label_tag }}
          {{ form.password2|add_class:"form-control" }}
        </div>
        <div class="row mb-3">
          <div class="col-md-6">
            {{ form.role.label_tag }}
            {{ form.role|add_class:"form-select" }}
          </div>
          <div class="col-md-6">
            {{ form.advisor.label_tag }}
            {{ form.advisor|add_class:"form-select" }}
          </div>
        </div>
        <button type="submit" class="btn btn-primary w-100">Register</button>
      </form>
    </div>
  </div>
</div>
{% endblock %}
```

Figure 39: Register View HTML Template

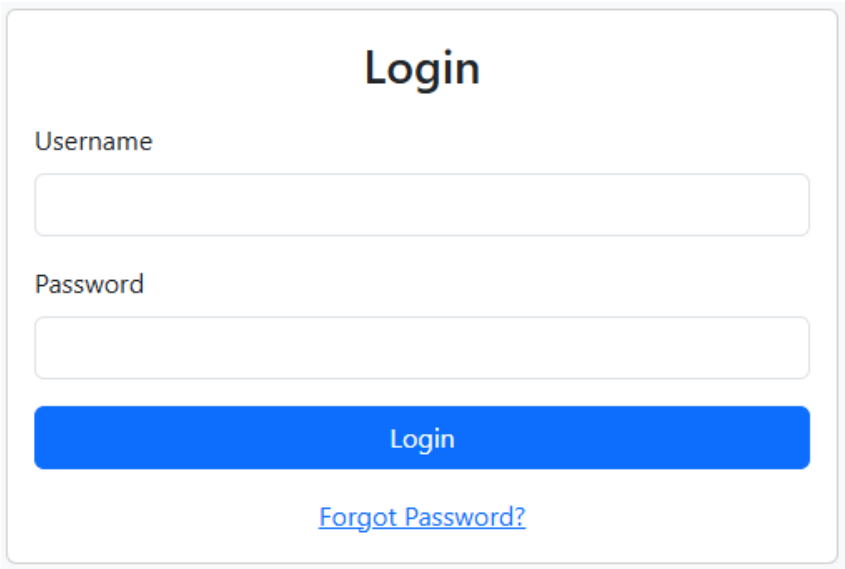

Figure 40: Login View Working Example

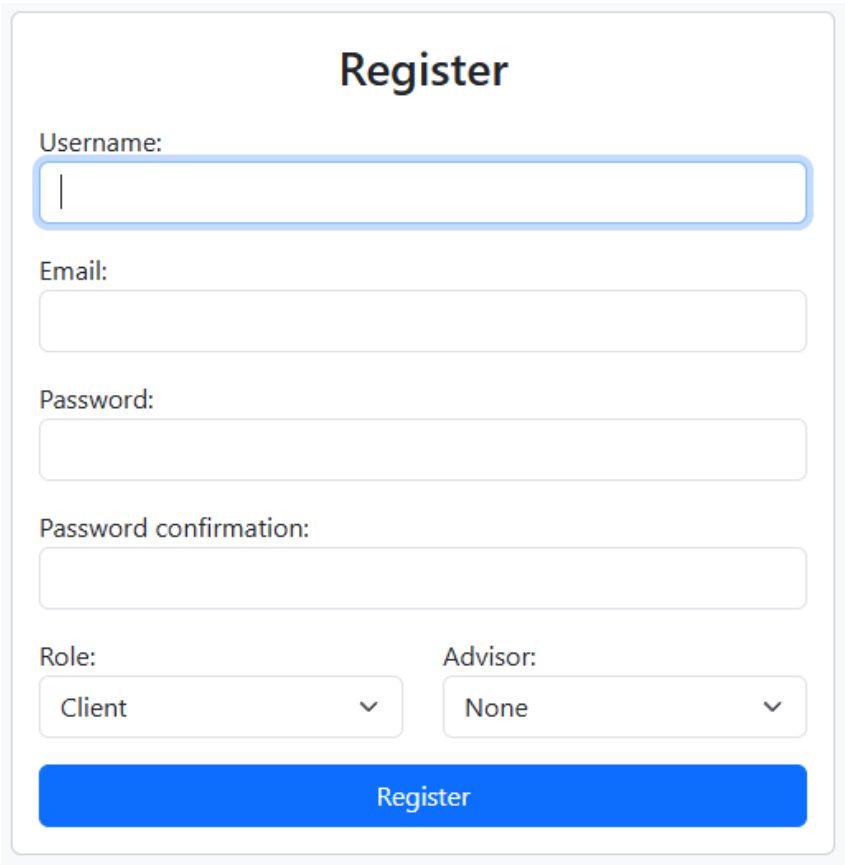

Figure 41: Register View Working Example

Dashboard and Stock Data Integration

The client dashboard displays real-time portfolio data, integrating live stock data via `yfinance` [Aroussi, 2025]. **Figures 42** and **43** show the view functions, while **Figures 44** and **45** show the HTML templates. Working examples can be seen in **Figures 46** and **47**.

```python
@login_required
def portfolio_view(request):
    portfolio, _ = Portfolio.objects.get_or_create(user=request.user)
    holdings = portfolio.holdings.all()
    user_balance = request.user.profile.balance
    total_holdings_value = 0
    holding_data = []

    for holding in holdings:
        price = holding.stock.last_price or 0
        value = float(holding.shares) * float(price)
        total_holdings_value += value
        holding_data.append({
            'holding': holding,
            'value': value,
            'percentage': 0,
        })

    total_portfolio_value = float(user_balance) + total_holdings_value
    cash_percentage = (float(user_balance) / total_portfolio_value * 100) if total_portfolio_value > 0 else 0

    # Build chart data
    chart_data = [{
        "label": "Cash",
        "value": float(user_balance),
        "percentage": cash_percentage
    }]
    for item in holding_data:
        pct = (item['value'] / total_portfolio_value * 100) if total_portfolio_value else 0
        item['percentage'] = pct
        chart_data.append({
            "label": item['holding'].stock.ticker,
            "value": item['value'],
            "percentage": pct
        })

    # Also fetch all stocks for the "Trade Stocks" dropdown
    stocks = Stock.objects.all().order_by('ticker')

    context = {
        'portfolio': portfolio,          #
        'holdings': holdings,
        'holding_data': holding_data,
        'chart_data': json.dumps(chart_data),
        'total_portfolio_value': total_portfolio_value,
        'cash_percentage': cash_percentage,
        'cash_balance': float(user_balance),
        'stocks': stocks,
    }
    return render(request, 'portfolio.html', context)
```

Figure 42: Portfolio View Function

```python
def stock_list_view(request):
    default_tickers = [
        "AAPL", "MSFT", "GOOGL", "AMZN", "TSLA","BRK-B", "JNJ", "V", "WMT", "JPM",
        "PG", "MA", "NVDA", "HD", "DIS", "BAC", "XOM", "VZ", "ADBE", "NFLX"
    ]
    current_count = Stock.objects.count()
    if current_count < 20:
        for ticker in default_tickers:
            try:
                yf_ticker = yf.Ticker(ticker)
                info = yf_ticker.info
                company_name = info.get('shortName') or info.get('longName') or ticker
                history = yf_ticker.history(period="1d")
                if not history.empty:
                    last_price = history['Close'].iloc[-1]
                else:
                    last_price = None

                Stock.objects.update_or_create(
                    ticker=ticker,
                    defaults={
                        'company_name': company_name,
                        'last_price': last_price,
                        'last_updated': timezone.now()
                    }
                )
            except Exception as e:
                print(f"Error updating {ticker}: {e}")
    stocks = Stock.objects.all().order_by('ticker')
    stock_data = []
    for stock in stocks:
        update_required = True
        if stock.last_updated:
            delta = timezone.now() - stock.last_updated
            if delta.total_seconds() < 60:
                update_required = False
        if update_required:
            try:
                yf_ticker = yf.Ticker(stock.ticker)
                history = yf_ticker.history(period="1d")
                if not history.empty:
                    last_price = history['Close'].iloc[-1]
                else:
                    last_price = stock.last_price
                stock.last_price = last_price
                stock.last_updated = timezone.now()
                stock.save()
            except Exception as e:
                print(f"Error updating {stock.ticker}: {e}")
                last_price = stock.last_price
        else:
            last_price = stock.last_price
        stock_data.append({
            'ticker': stock.ticker,
            'company_name': stock.company_name,
            'last_price': last_price,
        })
    return render(request, 'stock_list.html', {'stocks': stock_data})
```

Figure 43: Stock List View Function

```
{% extends "base.html" %}
{% block content %}
<div class="container my-5">
  <h1 class="text-center">Your Investment Portfolio</h1>
  <p class="text-center">Cash Balance: £{{ cash_balance|floatformat:2 }}</p>
  <h2>Your Holdings</h2>
  <table class="table table-striped">
    <thead><tr><th>Asset</th><th>Value</th><th>Percentage (%)</th></tr></thead>
    <tbody>
      <tr><td>Cash</td><td>£{{ cash_balance|floatformat:2 }}</td>
        <td>{{ cash_percentage|floatformat:2 }}%</td></tr>
      {% for item in holding_data %}
      <tr><td>{{ item.holding.stock.ticker }}</td><td>£{{ item.value|floatformat:2 }}</td>
        <td>{{ item.percentage|floatformat:2 }}%</td></tr>
      {% empty %}
      <tr><td colspan="3">No holdings found.</td></tr>
      {% endfor %}
    </tbody>
  </table>
  <div class="row mt-5">
    <div class="col-md-4 d-flex align-items-center justify-content-center">
      <div>
        <h2 class="mb-3 text-center">Portfolio Distribution</h2>
        <div style="display: flex; justify-content: center;">
          <canvas id="portfolioPieChart" style="width:300px; height:300px;"></canvas></div>
      </div>
    </div>
    <div class="col-md-4 d-flex align-items-center justify-content-center">
      <div>
        <h2 class="mb-3 text-center">Portfolio Value History</h2>
        <div style="display: flex; justify-content: center;">
          <canvas id="portfolioLineChart" style="width:300px; height:300px;"></canvas></div>
        <div id="portfolioChange" class="mt-2 text-center"></div>
      </div>
    </div>
    <div class="col-md-4 d-flex align-items-center justify-content-center">
      <div style="width:300px;">
        <h2 class="mb-3 text-center">Trade Stocks</h2>
        <form method="post" action="{% url 'client_transaction' %}">
          {% csrf_token %}
          <div class="mb-3">
            <label for="ticker" class="form-label">Stock Ticker</label>
            <select name="ticker" id="ticker" class="form-select" required>
              <option value="">-- Select Stock --</option>
              {% for stock in stocks %}<option value="{{ stock.ticker }}">
                {{ stock.ticker }} - {{ stock.company_name }}</option>{% endfor %}
            </select>
          </div>
          <div class="mb-3">
            <label for="shares" class="form-label">Number of Shares</label>
            <input type="number" name="shares" id="shares" class="form-control" step="0.01"
             placeholder="Enter number of shares" required>
          </div>
          <div class="d-flex justify-content-between">
            <button type="submit" name="action" value="buy" class="btn btn-primary">Buy</button>
            <button type="submit" name="action" value="sell" class="btn btn-danger">Sell</button>
          </div>
        </form>
      </div>
    </div>
  </div>
</div>
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
```

Figure 44: Portfolio View HTML Template

```
{% extends "base.html" %}
{% block content %}
<div class="container my-5">
  <h1 class="text-center mb-4">Available Stocks</h1>
  <div class="table-responsive">
    <table class="table table-striped">
      <thead class="table-dark">
        <tr>
          <th>Ticker</th>
          <th>Company Name</th>
          <th>Last Price</th>
          <th>Invest</th>
        </tr>
      </thead>
      <tbody>
        {% for stock in stocks %}
          <tr>
            <td>{{ stock.ticker }}</td>
            <td>{{ stock.company_name }}</td>
            <td>
              {% if stock.last_price %}
                £{{ stock.last_price|floatformat:2 }}
              {% else %}
                N/A
              {% endif %}
            </td>
            <td>
              <a href="{% url 'invest' stock.ticker %}" class="btn btn-primary">Invest</a>
            </td>
          </tr>
        {% empty %}
          <tr>
            <td colspan="4" class="text-center">No stocks available.</td>
          </tr>
        {% endfor %}
      </tbody>
    </table>
  </div>
</div>
{% endblock %}
```

Figure 45: Stock List HTML Template

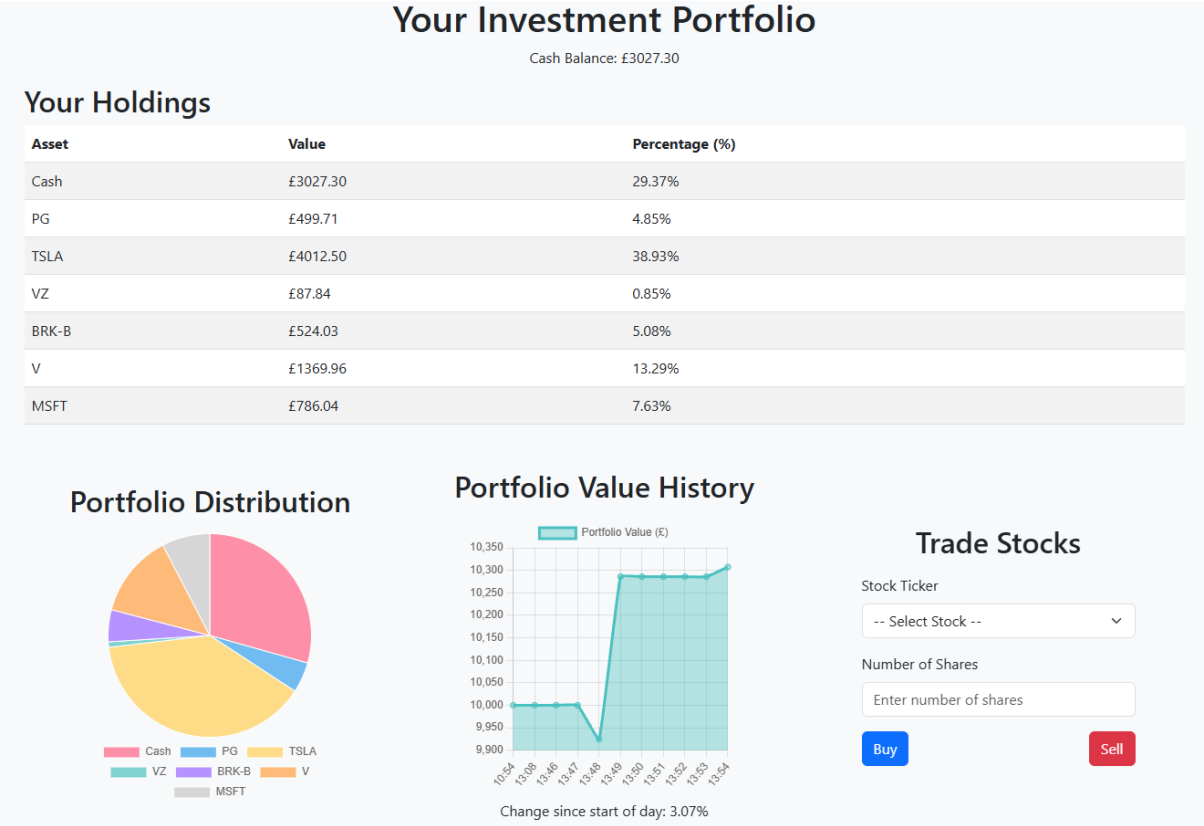

Figure 46: Portfolio View Working Example

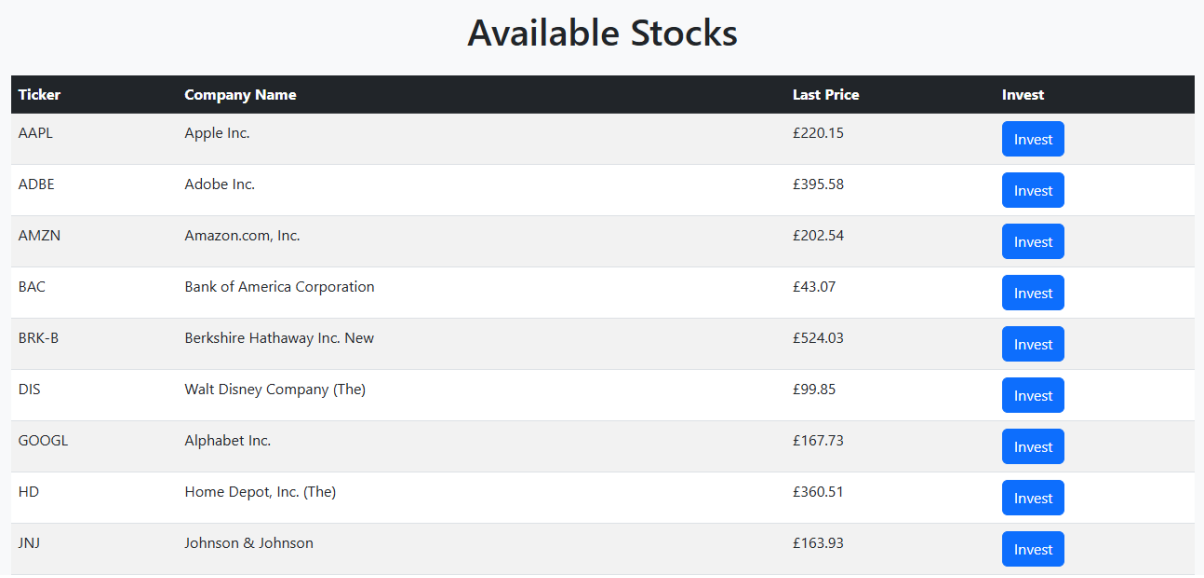

Figure 47: Stock List View Working Example

Purchase and Selling

Secure stock transactions update encrypted balances and holdings [PyCA, 2023]. **Figure 48** shows the transaction processing function, **Figure 49** displays the corresponding HTML template, and **Figure 50** provides a working example.

```python
@login_required
def client_transaction_view(request):
    if request.user.profile.role != 'client':
        messages.error(request, "ACCESS DENIED. Clients only.")
        return redirect('portfolio')
    portfolio, _ = Portfolio.objects.get_or_create(user=request.user)
    if request.method == 'POST':
        ticker = request.POST.get("ticker", "").strip().upper()
        shares_input = request.POST.get("shares")
        action = request.POST.get("action")
        try:
            shares = Decimal(shares_input)
        except Exception:
            messages.error(request, "Invalid share amount.")
            return redirect('portfolio')
        try:
            stock = Stock.objects.get(ticker=ticker)
        except Stock.DoesNotExist:
            messages.error(request, f"Stock '{ticker}' not found.")
            return redirect('portfolio')
        try:
            price = Decimal(str(yf.Ticker(stock.ticker).history(period="1d")['Close'].iloc[-1]))
        except Exception:
            messages.error(request, "Unable to retrieve stock data.")
            return redirect('portfolio')
        profile = request.user.profile
        timestamp = int(time.time())
        from .models import PQServerKey
        key_obj = PQServerKey.objects.filter(is_active=True).first()
        if action == 'buy':
            total_cost = shares * price
            if profile.balance < total_cost:
                messages.error(request, "Insufficient funds.")
                return redirect('portfolio')
            profile.balance -= total_cost; profile.save()
            holding, _ = Holding.objects.get_or_create(portfolio=portfolio, stock=stock)
            holding.shares += shares; holding.save()
            e_stock = encrypt_field(stock.ticker, b'investment_stock', b'investment stock encryption')
            e_type = encrypt_field('BUY', b'investment_type', b'investment type encryption')
            e_shares = encrypt_field(str(shares), b'investment_shares', b'investment shares encryption')
            e_price = encrypt_field(str(price), b'investment_price', b'investment price encryption')
            e_ts = encrypt_field(str(timestamp), b'investment_timestamp', b'investment timestamp encryption')
            InvestmentTransaction.objects.create(
                portfolio=portfolio,
                encrypted_stock=e_stock,
                encrypted_transaction_type=e_type,
                encrypted_shares=e_shares,
                encrypted_price=e_price,
                encrypted_timestamp=e_ts,
                key_used=key_obj
            )
            messages.success(request, f"Purchased {shares} shares of {ticker} at £{price:.2f}.")
        return redirect('portfolio')
    stocks = Stock.objects.all().order_by('ticker')
    return render(request, 'client_transaction.html', {'stocks': stocks, 'portfolio': portfolio})
```

Figure 48: Transaction Processing Function

```
{% extends "base.html" %}
{% block content %}
<div class="container my-5">
  <h1>Invest in {{ stock.ticker }}</h1>
  <p>Company: {{ stock.company_name }}</p>
  {% if stock.last_price %}
    <p>Current Price: £{{ stock.last_price|floatformat:2 }}</p>
  {% else %}
    <p>Current Price: N/A</p>
  {% endif %}
  {% if error %}
    <div class="alert alert-danger">{{ error }}</div>
  {% endif %}
  <form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit" class="btn btn-success">Buy Shares</button>
    <button type="submit" class="btn btn-danger">Sell Shares</button>
  </form>
</div>
{% endblock %}
```

Figure 49: Transaction HTML Template

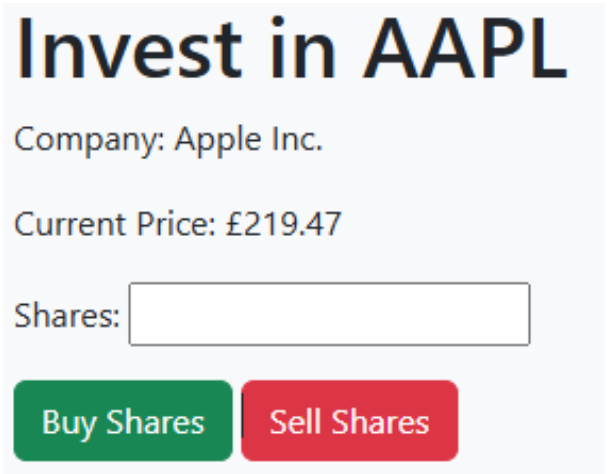

Figure 50: Working Transaction View Example

Advisor Dashboard

Advisors have a dedicated dashboard that aggregates client portfolio data and supports key functions: the advisor dashboard view function (**Figure 51**) aggregates client performance data, while the transaction view function (**Figure 52**) enables them to initiate and manage buy/sell orders. The client detail view (**Figure 53**) retrieves and displays detailed portfolio information, and the message view (**Figure 54**) facilitates secure, encrypted investment recommendations. The corresponding HTML templates for these functions are shown in **Figure 55** (dashboard), **Figure 56** (transaction interface), **Figure 57** (client details), and **Figure 58** (messaging), while **Figure 59** demonstrates full administrative integration and system functionality.

```python
@login_required
def advisor_view(request):
    # Only allow advisors.
    stocks = Stock.objects.all().order_by('ticker')
    if request.user.profile.role != 'advisor':
        messages.error(request, "Access denied.")
        return redirect('portfolio')
    client_profiles = Profile.objects.filter(role='client', advisor=request.user).order_by('user__username')
    clients_data = []
    total_cash = 0.0
    stock_totals = {}
    for client in client_profiles:
        portfolio, _ = Portfolio.objects.get_or_create(user=client.user)
        cash = float(portfolio.user.profile.balance)
        total_holdings_value = 0.0
        for holding in portfolio.holdings.all():
            price = float(holding.stock.last_price) if holding.stock.last_price else 0.0
            total_holdings_value += float(holding.shares) * price
        portfolio_value = cash + total_holdings_value
        total_cash += cash
        for holding in portfolio.holdings.all():
            ticker = holding.stock.ticker
            price = float(holding.stock.last_price) if holding.stock.last_price else 0.0
            if ticker in stock_totals:
                stock_totals[ticker]['shares'] += float(holding.shares)
            else:
                stock_totals[ticker] = {
                    'shares': float(holding.shares),
                    'price': price,
                    'company_name': holding.stock.company_name
                }
        clients_data.append({
            'user': client.user,
            'portfolio_value': portfolio_value,
        })
    chart_data = [{
        'label': "Cash",
        'value': total_cash
    }]
    for ticker, data in stock_totals.items():
        total_value = data['shares'] * data['price']
        chart_data.append({
            'label': ticker,
            'value': total_value,
            'shares': data['shares'],
            'company_name': data['company_name']
        })
    total_combined_value = total_cash + sum(item['value'] for item in chart_data if item['label'] != "Cash")
    for item in chart_data:
        item['percentage'] = (item['value'] / total_combined_value * 100) if total_combined_value > 0 else 0
    context = {
        'clients': clients_data,  # Now each client dict contains a 'portfolio_value' field.
        'stocks': stocks,
        'chart_data': json.dumps(chart_data),
    }
    return render(request, 'advisor.html', context)
```

Figure 51: Advisor Dashboard View Function

```python
@login_required
def advisor_transaction_view(request):
    if request.user.profile.role != 'advisor':
        messages.error(request, "Access denied.")
        return redirect('portfolio')
    if request.method == 'POST':
        client_id = request.POST.get("client")
        ticker = request.POST.get("ticker", "").strip().upper()
        shares_input = request.POST.get("shares")
        action = request.POST.get("action")
        try:
            shares = Decimal(shares_input)
        except Exception:
            messages.error(request, "Invalid share amount.")
            return redirect('advisor')
        try:
            client_profile = Profile.objects.get(user__id=client_id, role='client', advisor=request.user)
            client_user = client_profile.user
        except Profile.DoesNotExist:
            messages.error(request, "Client not found or is not assigned to you.")
            return redirect('advisor')
        portfolio, _ = Portfolio.objects.get_or_create(user=client_user)
        try:
            stock = Stock.objects.get(ticker=ticker)
        except Stock.DoesNotExist:
            messages.error(request, f"Stock '{ticker}' not found.")
            return redirect('advisor')
        try:
            yf_ticker = yf.Ticker(stock.ticker)
            history = yf_ticker.history(period="1d")
            if not history.empty:
                price = Decimal(str(history['Close'].iloc[-1]))
            else:
                messages.error(request, f"Stock price not available for {ticker}.")
                return redirect('advisor')
        except Exception:
            messages.error(request, "Error retrieving stock data.")
            return redirect('advisor')
        key_obj = PQServerKey.objects.filter(is_active=True).first()
        if not key_obj:
            messages.error(request, "No active encryption key found. Cannot record transaction.")
            return redirect('advisor')
        timestamp_str = str(int(time.time()))
        if action == 'buy':
            total_cost = shares * price
            if client_profile.balance < total_cost:
                messages.error(request, "Client has insufficient funds for this purchase.")
                return redirect('advisor')
            client_profile.balance -= total_cost; client_profile.save()
            holding, _ = Holding.objects.get_or_create(portfolio=portfolio, stock=stock, defaults={'shares': Decimal('0')})
            holding.shares += shares; holding.save()
            e_stock = encrypt_field(stock.ticker, b'investment_stock', b'investment stock encryption')
            e_type = encrypt_field('BUY', b'investment_type', b'investment type encryption')
            e_shares = encrypt_field(str(shares), b'investment_shares', b'investment shares encryption')
            e_price = encrypt_field(str(price), b'investment_price', b'investment price encryption')
            e_ts = encrypt_field(timestamp_str, b'investment_timestamp', b'investment timestamp encryption')
            InvestmentTransaction.objects.create(
                portfolio=portfolio,
                encrypted_stock=e_stock,
                encrypted_transaction_type=e_type,
                encrypted_shares=e_shares,
                encrypted_price=e_price,
                encrypted_timestamp=e_ts,
                key_used=key_obj,
            )
            AuditLog.objects.create(
                event=f"Advisor {request.user.username} bought {shares} shares of {ticker} at £{price:.2f} for client {client_user.username}.",
                user=request.user
            )
            messages.success(request, f"Bought {shares} shares of {ticker} for {client_user.username} at £{price:.2f} per share.")
```

Figure 52: Advisor Transaction View Function

```python
@login_required
def advisor_client_detail_view(request, client_id):
    # Only advisors can view client details.
    if request.user.profile.role != 'advisor':
        messages.error(request, "Access denied.")
        return redirect('portfolio')
    client_profile = get_object_or_404(Profile, id=client_id, role='client', advisor=request.user)
    # Get (or create) the client's portfolio
    portfolio, _ = Portfolio.objects.get_or_create(user=client_profile.user)
    holdings = portfolio.holdings.all()
    # Cash value
    cash = float(portfolio.user.profile.balance)
    # Initialize variables for total holding value
    total_holdings_value = 0.0
    holding_data = []
    # Calculate individual holding values and populate data for display
    for holding in holdings:
        if holding.stock.last_price:
            value = float(holding.shares) * float(holding.stock.last_price)
        else:
            value = 0.0
        total_holdings_value += value
        holding_data.append({
            'holding': holding,
            'value': value,
            'percentage': 0,  # will compute next
        })
    # Total Portfolio Value
    total_portfolio_value = cash + total_holdings_value
    # Compute cash percentage
    cash_percentage = (cash / total_portfolio_value * 100) if total_portfolio_value > 0 else 0
    # Update each holding's percentage
    for item in holding_data:
        item['percentage'] = (item['value'] / total_portfolio_value * 100) if total_portfolio_value > 0 else 0

    # Build chart data for pie chart (includes cash and each holding)
    chart_data = []
    chart_data.append({
        "label": "Cash",
        "value": cash,
        "percentage": cash_percentage
    })
    for item in holding_data:
        chart_data.append({
            "label": item['holding'].stock.ticker,
            "value": item['value'],
            "percentage": item['percentage']
        })
    context = {
        'client_profile': client_profile,
        'portfolio': portfolio,
        'holding_data': holding_data,
        'chart_data': json.dumps(chart_data),
        'total_portfolio_value': total_portfolio_value,
        'cash_percentage': cash_percentage,
    }
    return render(request, 'advisor_client_detail.html', context)
```

Figure 53: Advisor Client Detail View Function

```python
@login_required
def advisor_message_view(request):
    # Only advisors can send investment recommendations.
    if request.user.profile.role != 'advisor':
        messages.error(request, "Access denied.")
        return redirect('portfolio')

    if request.method == 'POST':
        recipient_username = request.POST.get("recipient").strip()
        message_text = request.POST.get("message").strip()

        if not recipient_username or not message_text:
            messages.error(request, "Both recipient and message are required.")
            return redirect('advisor_message')

        try:
            recipient = User.objects.get(username=recipient_username)
        except User.DoesNotExist:
            messages.error(request, "Client not found.")
            return redirect('advisor_message')

        # Encrypt the message using crypto_utils function.
        encrypted_text = crypto_utils.encrypt_message(message_text)
        Message.objects.create(
            sender=request.user,
            recipient=recipient,
            encrypted_text=encrypted_text
        )

        AuditLog.objects.create(
            event=f"Advisor {request.user.username} sent a recommendation to {recipient.username}.",
            user=request.user
        )

        messages.success(request, "Recommendation sent successfully.")
        # Redirect to the chat detail view for this recipient.
        return redirect('chat', username=recipient.username)

    # For GET requests, render the message form.
    return render(request, 'advisor_message.html')
```

Figure 54: Advisor Message View Function

```
<!-- Charts Section: Two Charts Side-by-Side -->
<div class="row mt-5">
  <!-- Pie Chart: Portfolio Distribution -->
  <div class="col-md-6 text-center">
    <h2 class="mb-3">Portfolio Distribution</h2>
    <div style="display: flex; justify-content: center;">
      <canvas id="portfolioPieChart" style="width:300px; height:300px;"></canvas>
    </div>
  </div>
  <!-- Line Chart: Portfolio Value History -->
  <div class="col-md-6 text-center">
    <h2 class="mb-3">Portfolio Value History</h2>
    <div style="display: flex; justify-content: center;">
      <canvas id="portfolioLineChart" style="width:300px; height:300px;"></canvas>
    </div>
    <div id="portfolioChange" class="mt-2"></div>
  </div>
</div>
```

Figure 55: Advisor Dashboard HTML Template

```
<!-- Combined Buy/Sell Transaction Form -->
<div class="col-md-6 d-flex justify-content-center">
  <div class="border rounded grid-card" style="height: 500px; width: 100%;">
    <h2 class="text-center">Buy/Sell Stocks for Client</h2>
    <form method="post" action="{% url 'advisor_transaction' %}">
      {% csrf_token %}
      <div class="mb-3">
        <label for="client" class="form-label">Select Client</label>
        <select name="client" id="client" class="form-select" required>
          <option value="">-- Select Client --</option>
          {% for client in clients %}
            <option value="{{ client.user.id }}">{{ client.user.username }}</option>
          {% endfor %}
        </select>
      </div>
      <div class="mb-3">
        <label for="ticker" class="form-label">Stock Ticker</label>
        <select name="ticker" id="ticker" class="form-select" required>
          <option value="">-- Select Stock --</option>
          {% for stock in stocks %}
            <option value="{{ stock.ticker }}">{{ stock.ticker }} - {{ stock.company_name }}</option>
          {% endfor %}
        </select>
      </div>
      <div class="mb-3">
        <label for="shares" class="form-label">Number of Shares</label>
        <input type="number" name="shares" id="shares" class="form-control" step="0.01" placeholder="Enter number of shares" required>
      </div>
      <div class="d-flex justify-content-between">
        <button type="submit" name="action" value="buy" class="btn btn-primary">Submit Purchase</button>
        <button type="submit" name="action" value="sell" class="btn btn-danger">Submit Sale</button>
      </div>
    </form>
  </div>
</div>
```

Figure 56: Advisor Transaction HTML Template

```
<!-- Client Portfolios Section -->
<div class="mb-5">
  <h2>Client Portfolios</h2>
  {% if clients %}
  <table class="table table-striped">
    <thead class="table-dark">
      <tr>
        <th>Client Username</th>
        <th>Portfolio Value (£)</th>
      </tr>
    </thead>
    <tbody>
      {% for client in clients %}
      <tr>
        <td>
          <a href="{% url 'advisor_client_detail' client.user.id %}">
            {{ client.user.username }}
          </a>
        </td>
        <td>£{{ client.portfolio_value|floatformat:2 }}</td>
      </tr>
      {% endfor %}
    </tbody>
  </table>
  {% else %}
    <p>No client portfolios found.</p>
  {% endif %}
</div>
```

Figure 57: Advisor Client Detail HTML Template

```
<!-- Investment Recommendation Form -->
<div class="col-md-6 d-flex justify-content-center">
  <div class="border rounded grid-card" style="height: 500px; width: 100%;">
    <h2 class="text-center">Send Investment Recommendation</h2>
    <form method="post" action="{% url 'advisor_message' %}">
      {% csrf_token %}
      <div class="mb-3">
        <label for="recipient_msg" class="form-label">Select Client</label>
        <select name="recipient" id="recipient_msg" class="form-select" required>
          <option value="">-- Select Client --</option>
          {% for client in clients %}
            <option value="{{ client.user.username }}">{{ client.user.username }}</option>
          {% endfor %}
        </select>
      </div>
      <div class="mb-3">
        <label for="message" class="form-label">Message</label>
        <textarea name="message" id="message" rows="5" class="form-control"
        placeholder="Enter your recommendation" required></textarea>
      </div>
      <div class="text-center">
        <button type="submit" class="btn btn-secondary">Send Recommendation</button>
      </div>
    </form>
  </div>
</div>
```

Figure 58: Advisor Message HTML Template

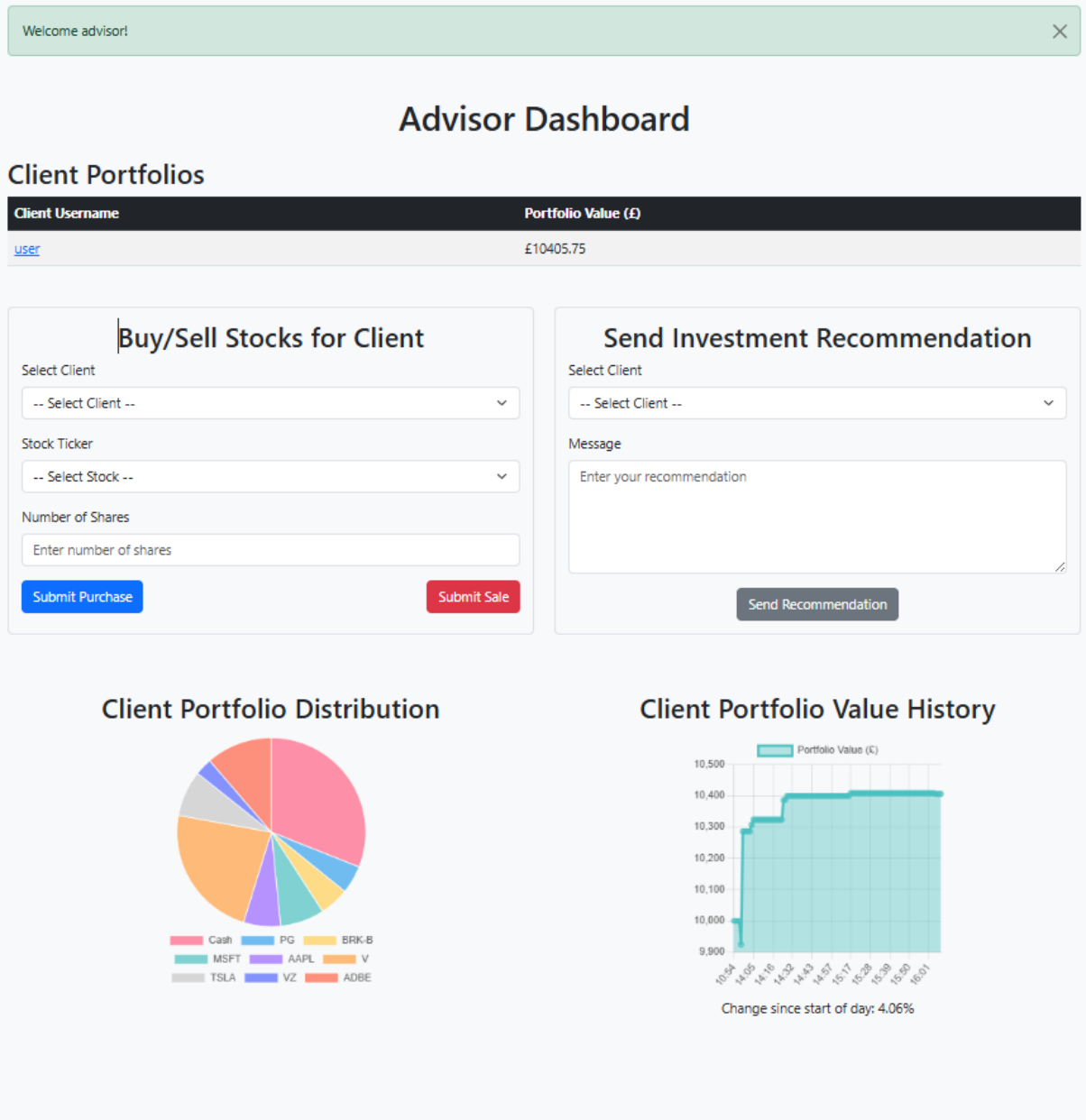

Figure 59: Final Working Advisor Dashboard

Admin Dashboard and Additional Features

The admin dashboard offers a comprehensive overview of system operations, including audit logs, transaction summaries, key management (with key rotations), and user management [Scarfone and Souppaya, 2006]. The admin dashboard view function (**Figure 60**) displays aggregated system status, while the delete user view (**Figure 61**) and create user view (**Figure 62**) facilitate account management. Detailed information on individual users is provided by the admin user detail view (**Figure 63**). The corresponding HTML templates for these functionalities are shown in **Figure 64**, **Figure 65**, **Figure 66**, and **Figure 67**, while the final working examples on the dashboard (**Figure 68**), delete user view (**Figure 69**), create user view (**Figure 70**), and user detail view (**Figure 71**) demonstrate the fully integrated admin functionality.

```python
@login_required
def admin_dashboard_view(request):
    if request.user.profile.role != 'admin':
        messages.error(request, "Access denied.")
        return redirect('portfolio')
    users = User.objects.all().order_by('username')
    audit_logs = AuditLog.objects.all().order_by('-timestamp')[:5]
    # Active server key for fallback
    algorithm, active_pub_key, active_server_priv = get_server_keys()
    server_public_key = base64.b64encode(active_pub_key).decode('utf-8')
    decrypted_transactions = []
    total_money = Decimal('0')
    valid_count = 0
    transactions = InvestmentTransaction.objects.all().order_by('-timestamp')
    for tx in transactions[:5]:
        try:
            # Use the correct key
            if tx.key_used:
                local_priv = base64.b64decode(tx.key_used.private_key)
            else:
                local_priv = active_server_priv
            decrypted_stock = decrypt_field(tx.encrypted_stock, b'investment_stock',
                                            b'investment stock encryption', local_priv)
            decrypted_type = decrypt_field(tx.encrypted_transaction_type, b'investment_type',
                                           b'investment type encryption', local_priv)
            shares_str = decrypt_field(tx.encrypted_shares, b'investment_shares',
                                       b'investment shares encryption', local_priv)
            price_str = decrypt_field(tx.encrypted_price, b'investment_price',
                                      b'investment price encryption', local_priv)
            shares = Decimal(shares_str)
            price = Decimal(price_str)
            total_money += shares * price
            valid_count += 1
            tx_display = {
                'id': tx.id,
                'stock': decrypted_stock,
                'transaction_type': decrypted_type,
                'shares': shares,
                'price': price,
                'timestamp': tx.timestamp.strftime("%Y-%m-%d %H:%M:%S")
            }
        except Exception as e:
            # If decryption fails, show placeholders
            tx_display = {
                'id': tx.id,
                'stock': "Error",
                'transaction_type': "Error",
                'shares': "Error",
                'price': "Error",
                'timestamp': tx.timestamp.strftime("%Y-%m-%d %H:%M:%S")
            }
        decrypted_transactions.append(tx_display)
    total_transactions = transactions.count()
    average_transaction = total_money / valid_count if valid_count else Decimal('0')
    context = {
        'users': users,
        'audit_logs': audit_logs,
        'recent_transactions': decrypted_transactions,
        'server_algorithm': algorithm,
        'server_public_key': server_public_key,
        'analytics': {
            'total_transactions': total_transactions,
            'total_money_moved': total_money,
            'average_transaction': average_transaction,
        },
    }
    return render(request, 'admin_dashboard.html', context)
```

Figure 60: Admin Dashboard View Implementation

```python
@login_required
def admin_user_delete_view(request, user_id):
    if request.user.profile.role != 'admin':
        messages.error(request, "Access denied.")
        return redirect('portfolio')
    user_obj = get_object_or_404(User, id=user_id)
    if request.method == 'POST':
        AuditLog.objects.create(
            event=f"User {user_obj.username} deleted by admin {request.user.username}.",
            user=request.user
        )
        user_obj.delete()
        messages.success(request, f"User {user_obj.username} deleted successfully.")
        return redirect('admin_dashboard')
    return render(request, 'admin_user_delete_confirm.html', {'user_obj': user_obj})
```

Figure 61: Admin Delete User View Implementation

```python
@login_required
def admin_create_user_view(request):
    if request.user.profile.role != 'admin':
        messages.error(request, "Access denied.")
        return redirect('portfolio')

    if request.method == 'POST':
        form = RegistrationForm(request.POST)
        if form.is_valid():
            user = form.save(commit=False)
            user.set_password(form.cleaned_data['password1'])
            user.save()  # This triggers the signal, which will create the Profile & Portfolio

            # set custom role & advisor from the form
            user.profile.role = form.cleaned_data.get('role')
            if user.profile.role == 'client':
                user.profile.advisor = form.cleaned_data.get('advisor')
            user.profile.save()

            AuditLog.objects.create(
                event=f"Admin {request.user.username} created user {user.username}.",
                user=request.user
            )
            messages.success(request, f"User {user.username} created successfully.")
            return redirect('admin_dashboard')
        else:
            for field in form:
                for error in field.errors:
                    messages.error(request, f"{field.label}: {error}")
            for error in form.non_field_errors():
                messages.error(request, error)
            return redirect('admin_create_user')
    else:
        form = RegistrationForm()
    return render(request, 'admin_create_user.html', {'form': form})
```

Figure 62: Admin Create User View Implementation

```python
@login_required
def admin_user_detail_view(request, user_id):
    if request.user.profile.role != 'admin':
        messages.error(request, "Access denied.")
        return redirect('portfolio')
    user_obj = get_object_or_404(User, id=user_id)
    profile = user_obj.profile
    portfolio, created = Portfolio.objects.get_or_create(user=user_obj)
    holdings = portfolio.holdings.all()
    cash = float(profile.balance)
    total_holdings_value = 0
    holding_data = []
    for holding in holdings:
        if holding.stock.last_price:
            value = float(holding.shares) * float(holding.stock.last_price)
        else:
            value = 0
        total_holdings_value += value
        holding_data.append({
            'holding': holding,
            'value': value,
            'percentage': 0,
        })
    total_portfolio_value = cash + total_holdings_value
    cash_percentage = (cash / total_portfolio_value * 100) if total_portfolio_value > 0 else 0
    for item in holding_data:
        item['percentage'] = ((item['value'] / total_portfolio_value) * 100) if total_portfolio_value > 0 else 0
    chart_data = [{
        "label": "Cash",
        "value": cash,
        "percentage": cash_percentage
    }]
    for item in holding_data:
        chart_data.append({
            "label": item['holding'].stock.ticker,
            "value": item['value'],
            "percentage": item['percentage']
        })
    extra_info = {}
    if profile.role == 'client':
        extra_info['advisor'] = profile.advisor
    elif profile.role == 'advisor':
        # For advisors, include the list of clients they manage.
        extra_info['clients_assigned'] = Profile.objects.filter(
            role='client', advisor=user_obj
        ).order_by('user__username')

    context = {
        'user_obj': user_obj,
        'portfolio': portfolio,
        'holding_data': holding_data,
        'chart_data': json.dumps(chart_data),
        'total_portfolio_value': total_portfolio_value,
        'cash_percentage': cash_percentage,
        'profile_role': profile.role,
        'cash': cash,
    }
    context.update(extra_info)
    return render(request, 'admin_user_detail.html', context)
```

Figure 63: Admin User Detail View Implementation

```
{% extends "base.html" %}
{% block content %}
<div class="container my-5">
  <h1 class="text-center mb-4">Admin Dashboard</h1>
  <p class="text-center">Hello, <strong>{{ user.username }}</strong>! You are logged in as <strong>Admin</strong>.</p>
  <!-- Analytics Section -->
  <div class="row text-center mb-4">
    <div class="col-md-4 mb-3">
      <div class="card bg-success text-white shadow">
        <div class="card-body">
          <h4 class="card-title">Total Transactions</h4>
          <p class="card-text display-5">{{ analytics.total_transactions }}</p>
        </div>
      </div>
    </div>
    <div class="col-md-4 mb-3">
      <div class="card bg-info text-white shadow">
        <div class="card-body">
          <h4 class="card-title">Total Money Moved (£)</h4>
          <p class="card-text display-5">£{{ analytics.total_money_moved|floatformat:2 }}</p>
        </div>
      </div>
    </div>
    <div class="col-md-4 mb-3">
      <div class="card bg-warning text-dark shadow">
        <div class="card-body">
          <h4 class="card-title">Average Transaction (£)</h4>
          <p class="card-text display-5">£{{ analytics.average_transaction|floatformat:2 }}</p>
        </div>
      </div>
    </div>
  </div>
  <!-- User Management Section -->
  <div class="mb-5">
    <div class="d-flex justify-content-between align-items-center">
      <h2>User Management</h2>
      <a href="{% url 'admin_create_user' %}" class="btn btn-primary">Create User</a>
    </div>
    <p>Total Users: {{ total_users }}</p>
    <div class="table-responsive">
      <table class="table table-bordered table-striped">
        <thead class="table-dark">
          <tr>
            <th>Username</th>
            <th>Email</th>
            <th>Role</th>
            <th>Actions</th>
          </tr>
        </thead>
        <tbody>
          {% for user in users %}
            <tr>
              <td><a href="{% url 'admin_user_detail' user.id %}">{{ user.username }}</a></td>
              <td>{{ user.email }}</td>
              <td>{{ user.profile.role }}</td>
              <td>
                <a href="{% url 'admin_user_detail' user.id %}" class="btn btn-sm btn-info">View</a>
                <a href="{% url 'admin_user_delete' user.id %}" class="btn btn-sm btn-danger">Delete</a>
              </td>
            </tr>
          {% empty %}
            <tr>
              <td colspan="4">No users found.</td>
            </tr>
          {% endfor %}
        </tbody>
      </table>
    </div>
  </div>
```

Figure 64: Admin Dashboard HTML Template

```
{% extends "base.html" %}
{% block content %}
<div class="container-fluid my-3" style="max-width: 1200px;">
  <h1>User Details for {{ user_obj.username }}</h1>
  <p>Email: {{ user_obj.email }}</p>
  <div class="row mt-3">
    <div class="col-md-6">
      <h2>Portfolio Distribution</h2>
      <p>Total Portfolio Value: £{{ total_portfolio_value|floatformat:2 }}</p>
      <table class="table table-striped">
        <thead>
          <tr><th>Asset</th><th>Value (£)</th><th>Percentage (%)</th></tr>
        </thead>
        <tbody>
          <tr><td>Cash</td><td>£{{ cash|floatformat:2 }}</td><td>{{ cash_percentage|floatformat:2 }}%</td></tr>
          {% for item in holding_data %}
          <tr><td>{{ item.holding.stock.ticker }}</td><td>£{{ item.value|floatformat:2 }}</td>
            <td>{{ item.percentage|floatformat:2 }}%</td></tr>
          {% empty %}
          <tr><td colspan="3">No holdings found.</td></tr>
          {% endfor %}
        </tbody>
      </table>
    </div>
    <div class="col-md-6 d-flex align-items-center justify-content-center">
      <canvas id="userPortfolioPieChart" style="width:300px; height:300px;"></canvas>
    </div>
  </div>
  <div class="row mt-3">
    <div class="col-md-6">
      {% if profile_role == "client" %}
        <h3>Advisor Information</h3>
        {% if advisor %}<p>Assigned Advisor: {{ advisor.username }}</p>{% else %}<p>No advisor assigned.</p>{% endif %}
      {% elif profile_role == "advisor" %}
        <h3>Assigned Clients</h3>
        {% if clients_assigned %}
          <ul class="list-group">{% for client in clients_assigned %}<li class="list-group-item">
            {{ client.user.username }}</li>{% endfor %}</ul>
        {% else %}<p>No clients assigned.</p>{% endif %}
      {% endif %}
    </div>
  </div>
  <div class="row mt-3">
    <div class="col">
      {% if user.profile.role == 'admin' %}
        <a href="{% url 'admin_dashboard' %}" class="btn btn-secondary">Back to Dashboard</a>
      {% elif user.profile.role == 'advisor' %}
        <a href="{% url 'advisor' %}" class="btn btn-secondary">Back to Dashboard</a>
      {% else %}
        <a href="{% url 'portfolio' %}" class="btn btn-secondary">Back to Dashboard</a>
      {% endif %}
    </div>
  </div>
</div>
```

Figure 65: Admin Delete User HTML Template

```
{% extends "base.html" %}
{% load widget_tweaks %}
{% block content %}
<div class="container d-flex justify-content-center align-items-center" style="min-height: 80vh;">
  <div class="card" style="width: 100%; max-width: 500px;">
    <div class="card-body">
      <h3 class="card-title mb-3 text-center">Create New User</h3>
      <form method="POST">
        {% csrf_token %}
        <div class="mb-3">
          {{ form.username.label_tag }}
          {{ form.username|add_class:"form-control" }}
        </div>
        <div class="mb-3">
          {{ form.email.label_tag }}
          {{ form.email|add_class:"form-control" }}
        </div>
        <div class="mb-3">
          {{ form.password1.label_tag }}
          {{ form.password1|add_class:"form-control" }}
        </div>
        <div class="mb-3">
          {{ form.password2.label_tag }}
          {{ form.password2|add_class:"form-control" }}
        </div>
        <div class="row mb-3">
          <div class="col-md-6">
            {{ form.role.label_tag }}
            {{ form.role|add_class:"form-select" }}
          </div>
          <div class="col-md-6">
            {{ form.advisor.label_tag }}
            {{ form.advisor|add_class:"form-select" }}
          </div>
        </div>
        <button type="submit" class="btn btn-primary w-100">Create User</button>
      </form>
    </div>
  </div>
</div>
{% endblock %}
```

Figure 66: Admin Create User HTML Template

```
{% extends "base.html" %}
{% block content %}
<div class="container-fluid my-3" style="max-width: 1200px;">
  <h1>User Details for {{ user_obj.username }}</h1>
  <p>Email: {{ user_obj.email }}</p>
  <div class="row mt-3">
    <div class="col-md-6">
      <h2>Portfolio Distribution</h2>
      <p>Total Portfolio Value: £{{ total_portfolio_value|floatformat:2 }}</p>
      <table class="table table-striped">
        <thead>
          <tr><th>Asset</th><th>Value (£)</th><th>Percentage (%)</th></tr>
        </thead>
        <tbody>
          <tr><td>Cash</td><td>£{{ cash|floatformat:2 }}</td><td>{{ cash_percentage|floatformat:2 }}%</td></tr>
          {% for item in holding_data %}
          <tr><td>{{ item.holding.stock.ticker }}</td><td>£{{ item.value|floatformat:2 }}</td>
            <td>{{ item.percentage|floatformat:2 }}%</td></tr>
          {% empty %}
          <tr><td colspan="3">No holdings found.</td></tr>
          {% endfor %}
        </tbody>
      </table>
    </div>
    <div class="col-md-6 d-flex align-items-center justify-content-center">
      <canvas id="userPortfolioPieChart" style="width:300px; height:300px;"></canvas>
    </div>
  </div>
  <div class="row mt-3">
    <div class="col-md-6">
      {% if profile_role == "client" %}
        <h3>Advisor Information</h3>
        {% if advisor %}<p>Assigned Advisor: {{ advisor.username }}</p>{% else %}<p>No advisor assigned.</p>{% endif %}
      {% elif profile_role == "advisor" %}
        <h3>Assigned Clients</h3>
        {% if clients_assigned %}
          <ul class="list-group">{% for client in clients_assigned %}<li class="list-group-item">
            {{ client.user.username }}</li>{% endfor %}</ul>
        {% else %}<p>No clients assigned.</p>{% endif %}
      {% endif %}
    </div>
  </div>
  <div class="row mt-3">
    <div class="col">
      {% if user.profile.role == 'admin' %}
        <a href="{% url 'admin_dashboard' %}" class="btn btn-secondary">Back to Dashboard</a>
      {% elif user.profile.role == 'advisor' %}
        <a href="{% url 'advisor' %}" class="btn btn-secondary">Back to Dashboard</a>
      {% else %}
        <a href="{% url 'portfolio' %}" class="btn btn-secondary">Back to Dashboard</a>
      {% endif %}
    </div>
  </div>
</div>
```

Figure 67: Admin User Detail HTML Template

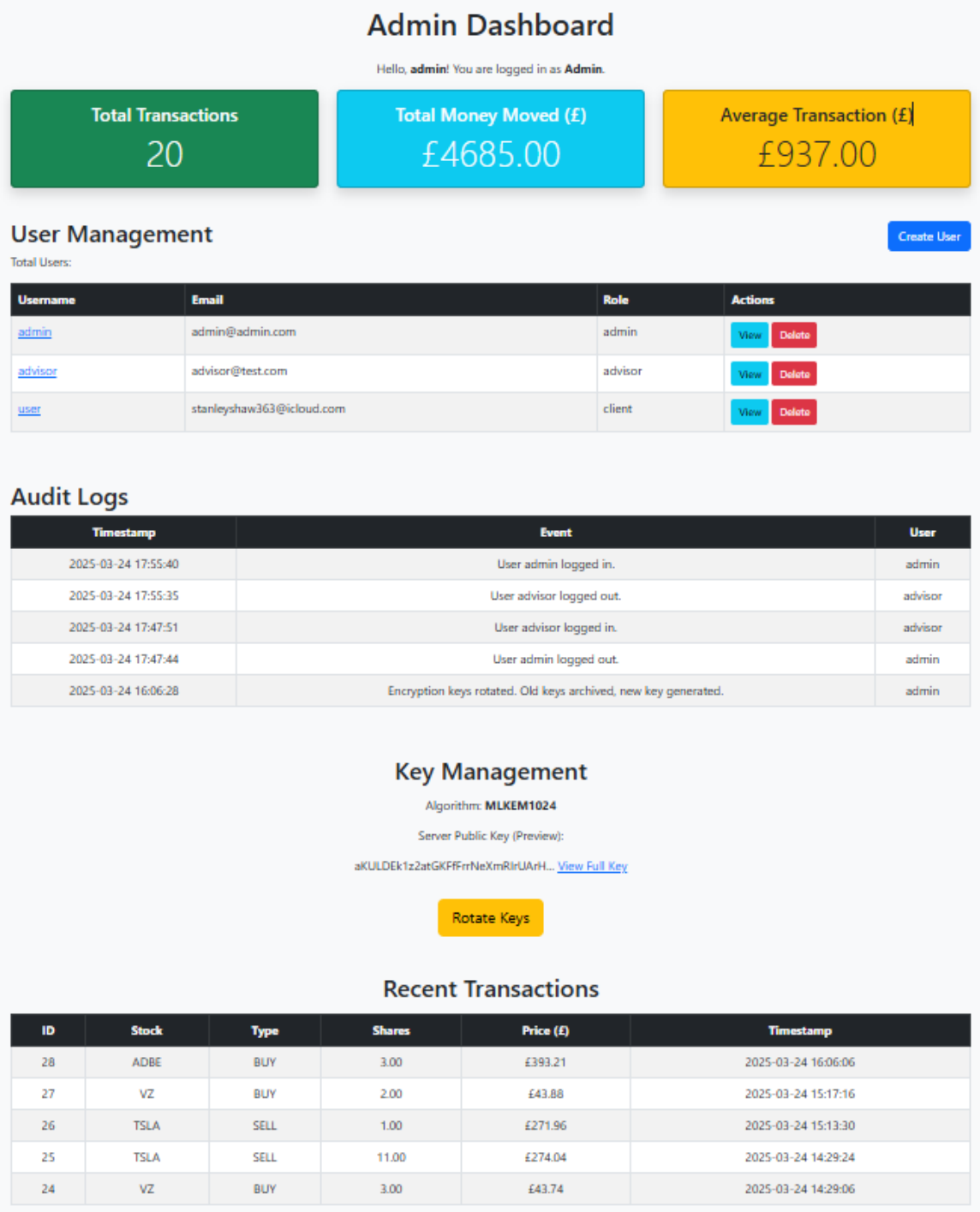

Figure 68: Final Working Admin Dashboard

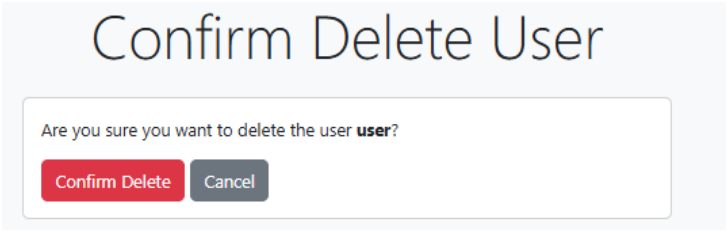

Figure 69: Admin Delete User View Working Example

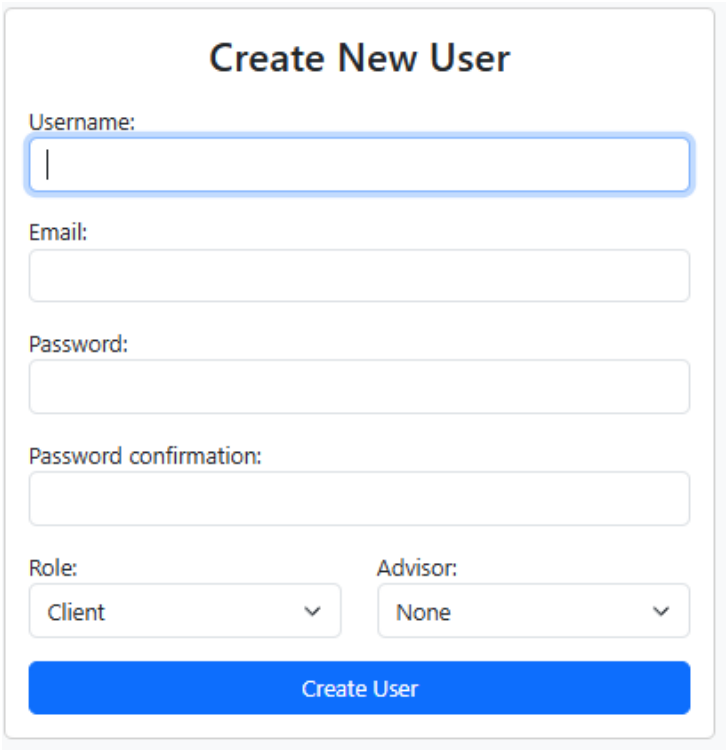

Figure 70: Admin Create User View Working Example

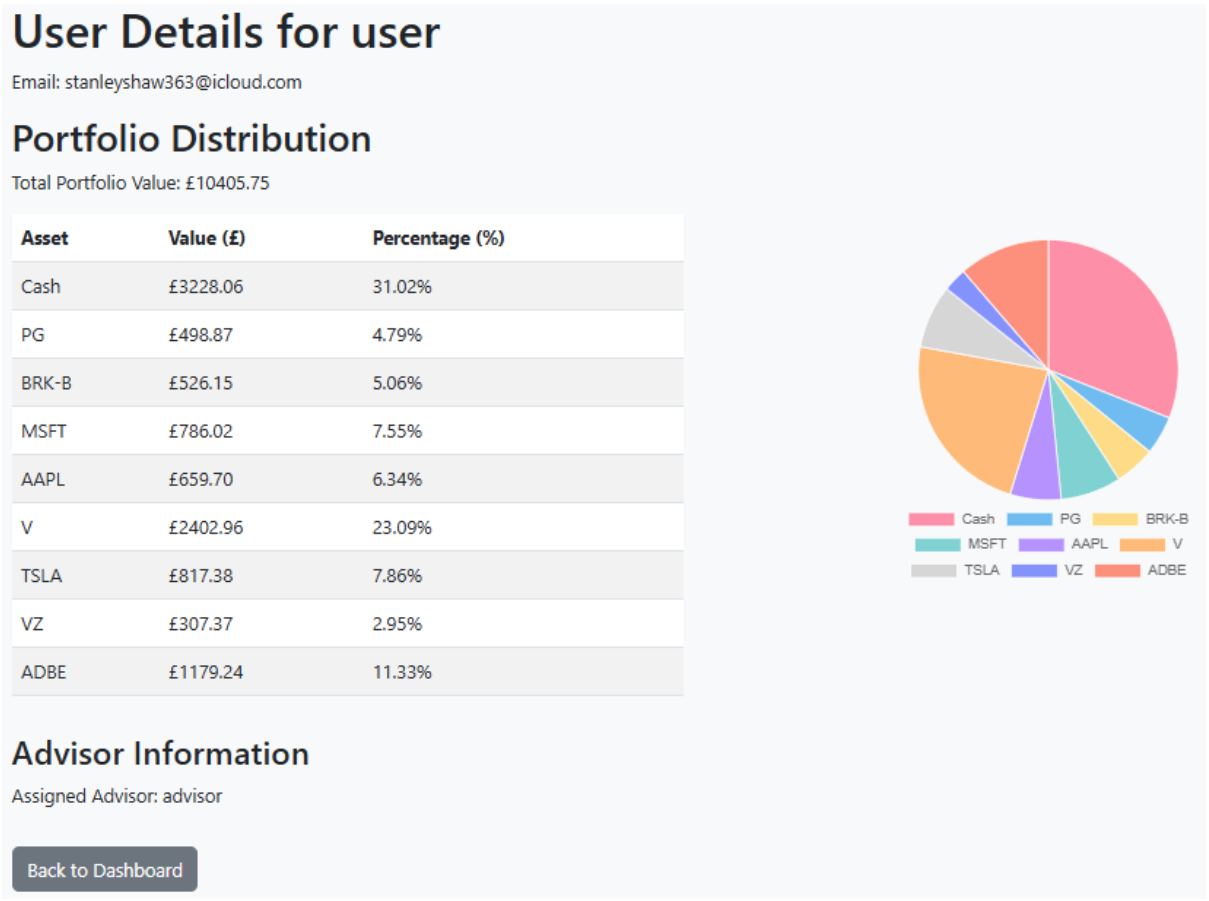

Figure 71: Admin User Detail View Working Example

4 Security Analysis

The cryptosystem for MyFinance Inc. has been designed with multiple layers of security controls to ensure confidentiality, integrity, and authenticity of financial data. This section outlines the main threats facing the system and highlights the mechanisms in place to mitigate them through the use of sequence diagrams.

4.1 Replay Attacks

A replay attack occurs when an attacker intercepts a valid message and maliciously re-sends it to trick the system. In this cryptosystem, every encryption operation includes a `nonce`—a 96-bit number generated using `os.urandom(12)`. This ensures that each encryption operation is unique. AES-GCM uses this nonce to produce different ciphertexts even when the same plaintext is encrypted multiple times. Since ciphertexts become non-deterministic, any replay attempts will result in authentication failure, rendering replay attacks ineffective [Stallings, 2017] (see **Figure 72**).

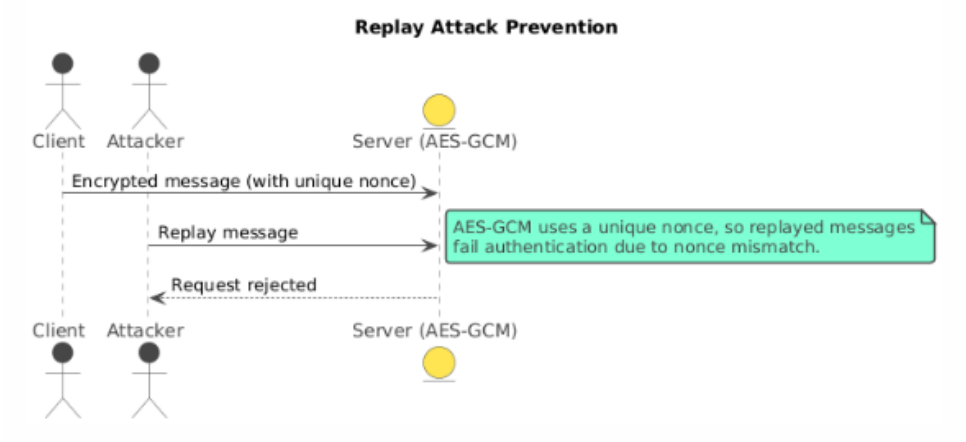

Figure 72: Replay Attack Example

4.2 Man-in-the-Middle (MitM) Attacks

MitM attacks occur when an adversary intercepts communication between two legitimate parties. MyFinance mitigates this at multiple levels, as all communication between clients and the server is protected using HTTPS, enforced by Django's `SECURE_SSL_REDIRECT` and HSTS headers [Rescorla, 2018]. At the cryptographic level, the key exchange protocol uses MLKEM-1024 (a post-quantum secure KEM) [NIST, 2024], making it resistant even to powerful quantum-enabled adversaries. AES-GCM provides authenticated encryption, so any tampered messages will fail decryption, even if successfully intercepted [Dworkin, 2007] (see **Figure 73**).

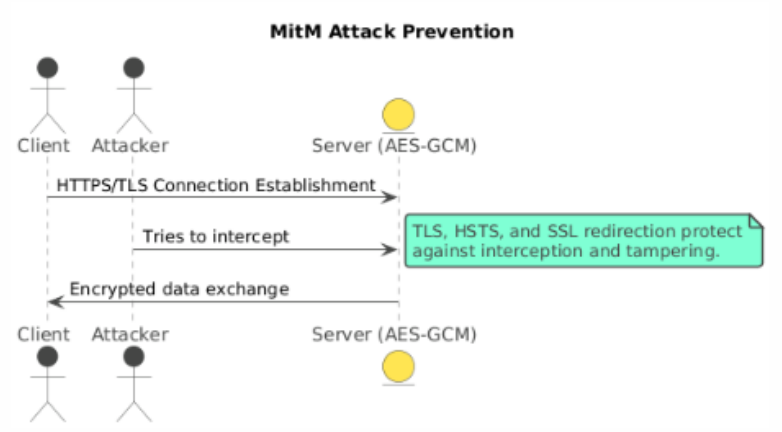

Figure 73: MitM Attack Example

4.3 Harvest-Now, Decrypt-Later Attacks (Post-Quantum Threats)

This emerging threat is based on the idea that encrypted data captured today may be decrypted in the future by a quantum computer. To protect against this, the system uses the NIST finalist algorithm MLKEM-1024 for key encapsulation [NIST, 2024]. By using a quantum-resistant scheme from the outset, even future adversaries equipped with quantum technology will not be able to break historical encrypted data [Chen et al., 2016]. This design future-proofs the application's cyber security (see **Figure 74**).

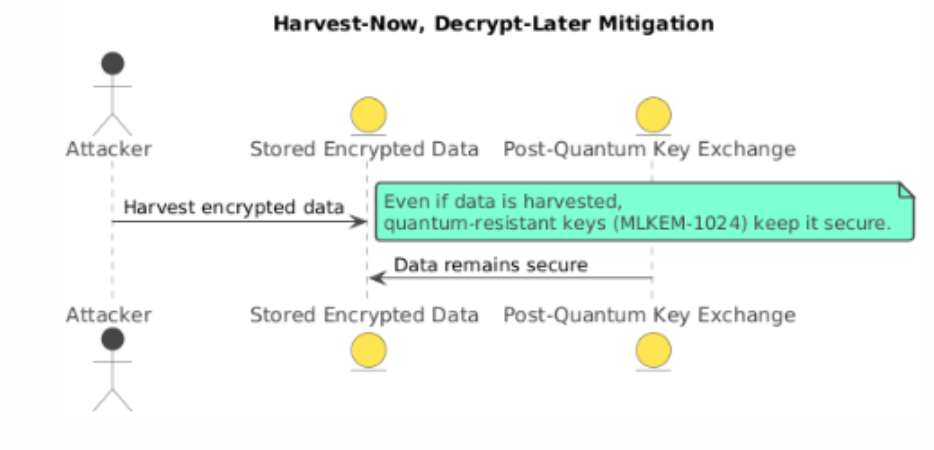

Figure 74: Harvest-Now, Decrypt-Later Attack Example

4.4 Data Breaches and Unauthorised Data Access

The application assumes that data breaches are highly feasible due to emerging technologies and undiscovered bugs; this is mitigated through the use of symmetric encryption. User balances are encrypted using AES-256 with per-field key derivation (HKDF) [NIST, 2001, Krawczyk and Eronen, 2010]. Investment transactions, stock tickers, transaction types, share amounts, and prices are individually encrypted using context-specific salts and info strings. Each encrypted value includes the ID of the public key used, so even archived data can be decrypted securely after key rotation. Moreover, access to user data is also controlled via Django's role-based permissions [Sandhu et al., 1996], ensuring that clients, advisors, and administrators can only access authorised data in accordance with their roles [Django, 2023] (see **Figure 75**). Additionally, industry reports highlight that data breaches are one of the most common security incidents [Verizon, 2021].

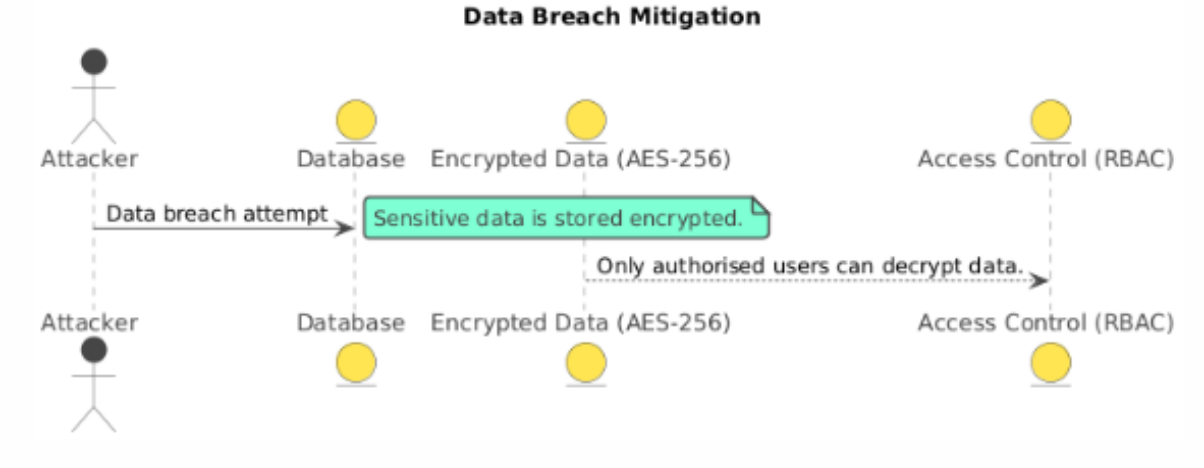

Figure 75: Data Breach Example

4.5 Brute Force and Credential Attacks

Brute force and credential attacks involve an adversary attempting to guess passwords through systematic trial and error or by leveraging high-performance hardware such as GPUs or ASICs to accelerate password guessing. To mitigate these threats, the system enforces strong password policies through rigorous validators that prevent the use of weak credentials [OWASP, 2025c]. Furthermore, passwords are hashed using a custom-configured Argon2 implementation configured with increased memory and time parameters, which significantly increases the computational effort required for brute force attacks [Biryukov et al., 2016] (see **Figure 76**). In addition, research has shown that multi-factor authentication can further reduce the risks associated with credential attacks [Bonneau et al., 2012].

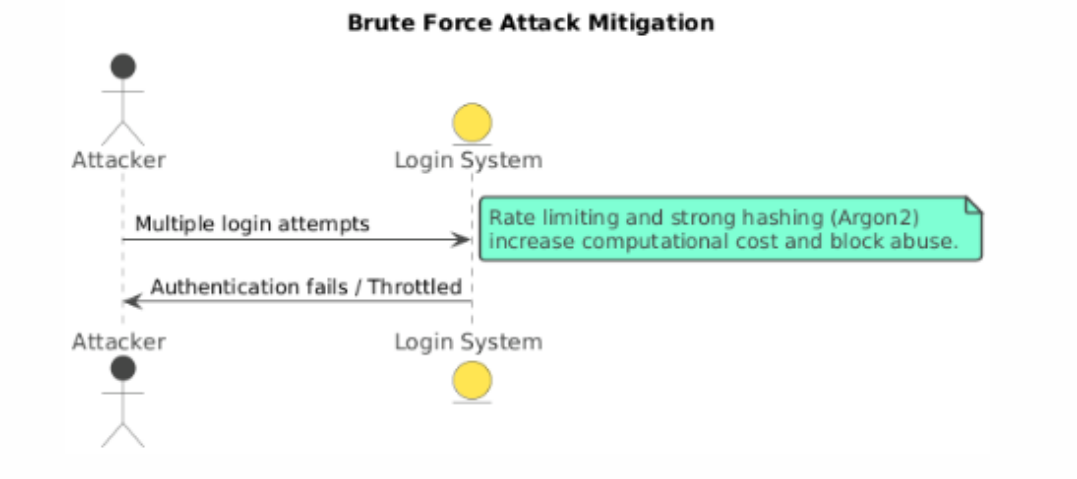

Figure 76: Brute Force Attack Example

4.6 Tampering and Integrity Violations

All encrypted fields use AES-GCM, which includes a built-in authentication tag. Any alteration of the ciphertext, even by a single bit, results in decryption failure [Dworkin, 2007]. This ensures the integrity of encrypted financial data such as balances, transactions, and messages. Additionally, for secure transaction messaging, the system could be extended to include digital signatures using authenticated ephemeral keys (already included in `crypto_utils.py`) [Menezes and Vanstone, 1997] (see **Figure 77**).

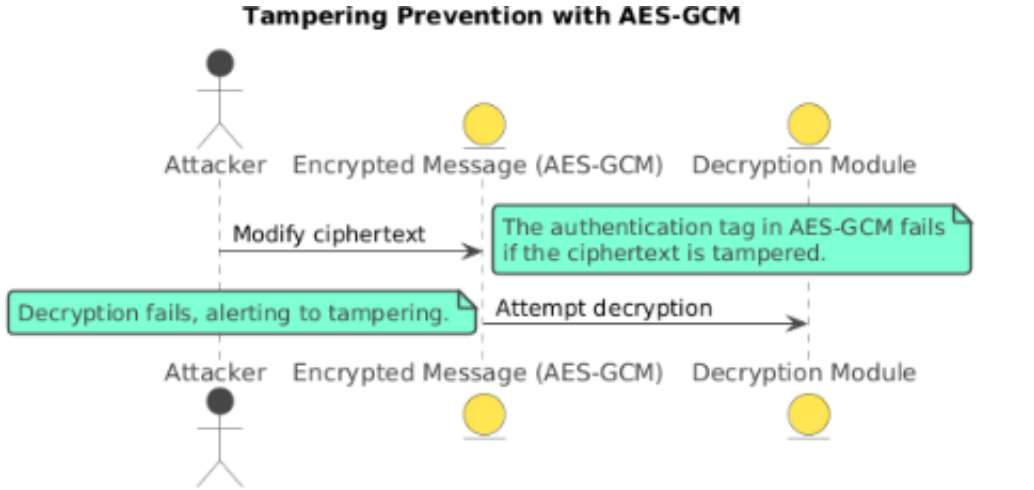

Figure 77: Tampering Attack Example

4.7 Key Compromise and Key Management Weaknesses

If a cryptographic key is compromised, all data encrypted with that key is immediately at risk. To reduce this threat, the system employs robust key management strategies, including a key rotation mechanism that allows administrators to securely deactivate old keys and generate new ones using MLKEM-1024 [Barker, 2020, NIST, 2024], thereby reducing the window of vulnerability. Every encrypted field includes a reference to the key used, ensuring that historical data remains decryptable even after key rotation; and the use of a cached key retrieval function enhances performance while limiting unnecessary handling of key material [Boneh and Shoup, 2020] (see **Figure 78**).

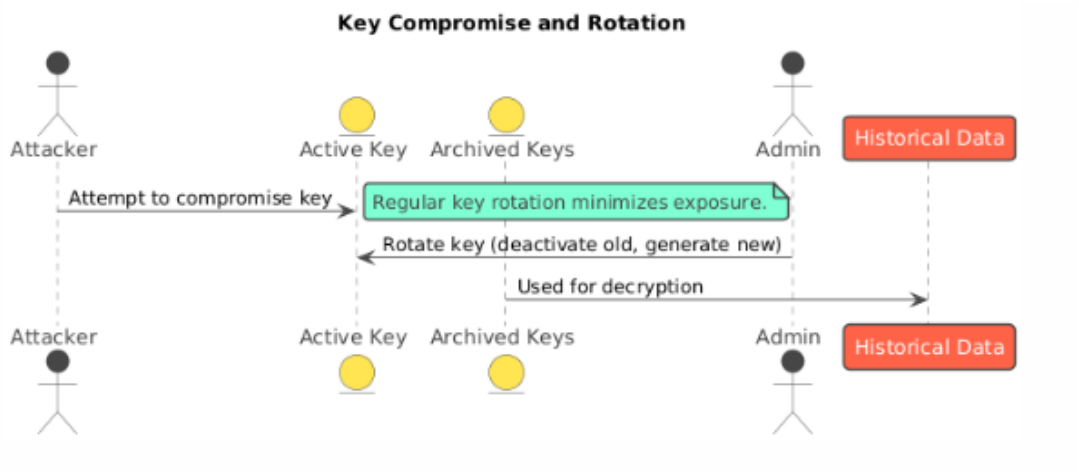

Figure 78: Key Compromise Mitigations Example

4.8 Privilege Escalation

Privilege escalation is a security vulnerability where a user gains unauthorised access to higher-level permissions or functionality. In order to prevent this, role-based access control (RBAC) is strictly enforced [Sandhu et al., 1996, Ferraiolo et al., 2003]. Only users with the admin role (set through the superuser flag) can perform sensitive administrative operations, including key rotation, user management, and audit log access. Advisors can view client portfolios, but cannot act outside their assigned clients. Clients cannot access admin or advisor functionality. These restrictions are enforced both in the UI and at the view-level using decorators and role checks [Django, 2023] (see **Figure 79**).

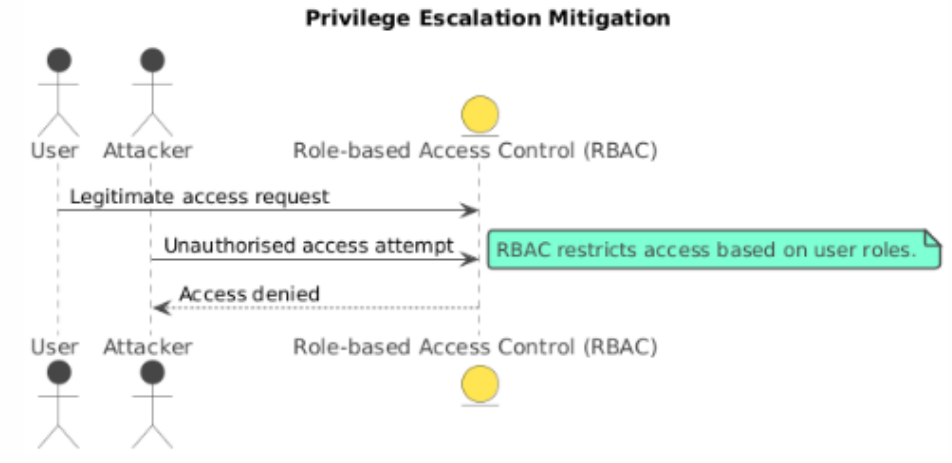

Figure 79: Privilege Escalation Example

4.9 Insider Threats and Auditability

The system incorporates auditing of sensitive operations. Actions like registration, login, transaction execution, and message sending are logged using the `AuditLog` model. This log allows administrators to trace potentially malicious or accidental misuse of the system and provides accountability for all users, including insiders [Scarfone and Souppaya, 2006, OWASP, 2025a]. Research indicates that proper audit logging is essential for detecting insider threats [Böhme, 2012] (see **Figure 80**).

Figure 80: Insider Threat Mitigation Example

4.10 Message Interception and Spoofing

Message interception and spoofing involve attackers eavesdropping on or manipulating communications between users. To mitigate this, all user-to-user messages (such as advisor recommendations) are encrypted using AES-GCM [Dworkin, 2007, Rescorla, 2018]. This ensures confidentiality, prevents unauthorised reading, even by database administrators, and detects any tampering through built-in authentication mechanisms [Menezes and Vanstone, 1997] (see **Figure 81**).

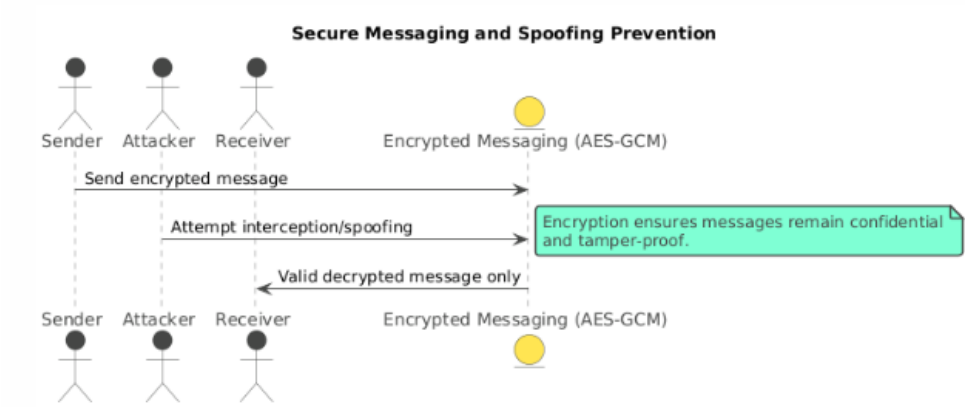

Figure 81: Message Interception/Spoofing Mitigation Example

4.11 Denial-of-Service and Abuse

Distributed Denial-of-Service (DDoS) attacks overwhelm a target by flooding it with excessive traffic from numerous compromised sources, rendering the target unable to respond to legitimate requests. The system includes explicit rate limiting and connection restrictions. Specifically, rate limiting is used to restrict login attempts or form submissions, ensuring that excessive requests are blocked, while connection restrictions help enforce that only connections from certain approved IP addresses are allowed [Mirkovic and Reiher, 2004]. This approach is further supported by modern web frameworks like Django [Django, 2023] (see **Figure 82**).

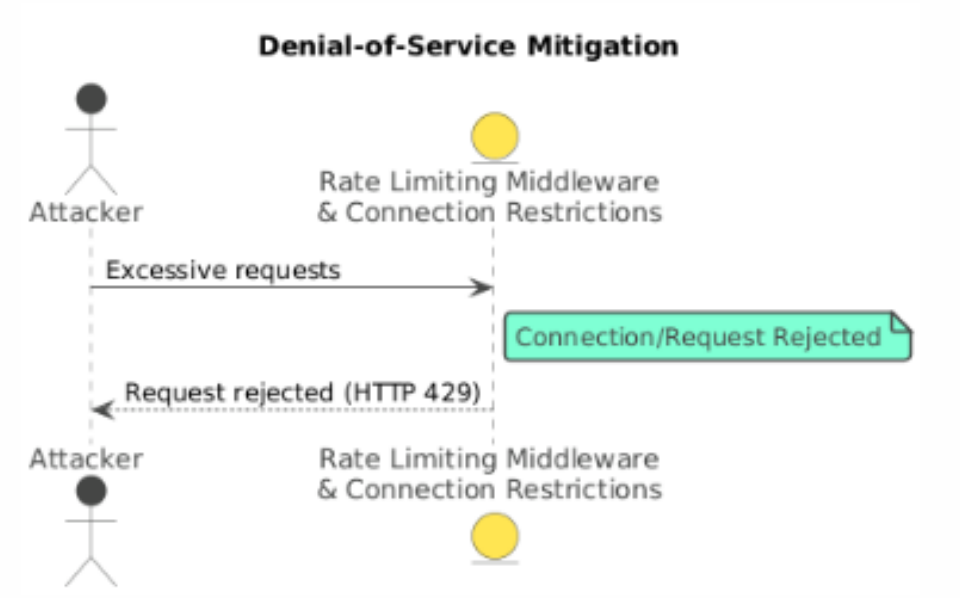

Figure 82: DDOS Mitigation Example

5 Testing and Evaluation

Extensive testing was performed on all components of the system to ensure it met the required security and functionality criteria. The testing process involved unit tests for individual cryptographic functions, integration tests for user operations and role-based functionalities, as well as performance and scalability assessments [Myers et al., 2011, Beizer, 1995].

5.1 Unit Testing

This section highlights key unit tests that verify the correctness, robustness, and edge-case handling of the cryptographic and hashing functions [Myers et al., 2011, Beizer, 1995]. Figures **83**–**90** show code snippets of the tests and their intended purposes. **Figure 83** shows my custom Argon2 hasher test, ensuring that valid passwords are accepted and invalid ones are rejected.

```python
class Argon2HasherTests(TestCase):
    def test_argon2_password_hashing(self):
        from django.contrib.auth.hashers import make_password, check_password
        password = "SuperSecurePassword123!"
        hashed_password = make_password(password)
        self.assertTrue(check_password(password, hashed_password))
        self.assertFalse(check_password("WrongPassword", hashed_password))
```

Figure 83: `Argon2HasherTests` verifying that the hashing works

Figure 84 highlights tests for encrypting unusually large balances, handling zero-value balances, and verifying that tampered ciphertext or repeated nonces cause decryption errors.

```python
# Balance Encryption/Decryption for Large Values
def test_encrypt_decrypt_large_balance(self):
    balance = Decimal("9999999999.99")
    encrypted = crypto_utils.encrypt_balance(balance)
    decrypted = crypto_utils.decrypt_balance(encrypted)
    self.assertEqual(balance, decrypted)

# Balance Encryption/Decryption for Zero Value
def test_encrypt_balance_zero(self):
    balance = Decimal("0.00")
    encrypted = crypto_utils.encrypt_balance(balance)
    decrypted = crypto_utils.decrypt_balance(encrypted)
    self.assertEqual(balance, decrypted)

# Nonce Randomness: Same input produces different encrypted outputs.
def test_encryption_produces_different_outputs_for_same_input(self):
    message = "duplicate input"
    encrypted1 = crypto_utils.encrypt_message(message)
    encrypted2 = crypto_utils.encrypt_message(message)
    self.assertNotEqual(encrypted1, encrypted2)

# Decrypt Field with Wrong Salt
def test_decrypt_field_with_wrong_salt_fails(self):
    value = "sensitive field"
    salt = b'correct_salt'
    info = b'field'
    encrypted = crypto_utils.encrypt_field(value, salt, info)
    with self.assertRaises(Exception):
        crypto_utils.decrypt_field(encrypted, b'wrong_salt', info)

# Decrypt Field with Wrong Info
def test_decrypt_field_with_wrong_info_fails(self):
    value = "sensitive field"
    salt = b'salt'
    encrypted = crypto_utils.encrypt_field(value, salt, b'correct_info')
    with self.assertRaises(Exception):
        crypto_utils.decrypt_field(encrypted, salt, b'wrong_info')
```

Figure 84: Unit Tests covering edge cases

Figure 85 illustrates more edge cases: attempting decryption with the wrong salt or info, missing PQServerKey records, corrupt ciphertext, and encrypting/decrypting Unicode data.

```python
# Decryption with Missing Key
def test_decrypt_balance_with_missing_key_raises_exception(self):
    from .models import PQServerKey
    balance = Decimal("100.00")
    encrypted = crypto_utils.encrypt_balance(balance)
    # Delete all keys so that the key used for encryption is missing.
    PQServerKey.objects.all().delete()
    with self.assertRaises(Exception):
        crypto_utils.decrypt_balance(encrypted)

# Corrupt Encrypted Data
def test_decrypt_message_with_corrupted_data_raises_exception(self):
    message = "Hello"
    encrypted = crypto_utils.encrypt_message(message)
    # Tamper with the encrypted message.
    tampered = encrypted[:-5] + "abcde"
    with self.assertRaises(Exception):
        crypto_utils.decrypt_message(tampered)

# Field Encryption/Decryption with Unicode Data
def test_encrypt_decrypt_field_unicode(self):
    value = "sëcrët🚀data"
    salt = b'field_salt'
    info = b'unicode'
    encrypted = crypto_utils.encrypt_field(value, salt, info)
    decrypted = crypto_utils.decrypt_field(encrypted, salt, info)
    self.assertEqual(decrypted, value)
```

Figure 85: Additional edge-case Unit tests

Figure 86 shows how I validate that newly generated keys become active and that the cryptographic algorithm matches my "MLKEM1024" scheme.

```python
# Key Management
def test_generate_key_creates_active_key(self):
    key = crypto_utils.generate_new_key()
    self.assertTrue(key.is_active)
    self.assertEqual(key.algorithm, "MLKEM1024")

def test_get_server_keys_returns_same_key_if_cached(self):
    key1 = crypto_utils.get_server_keys()
    key2 = crypto_utils.get_server_keys()
    self.assertEqual(key1, key2)
```

Figure 86: Key Management Unit tests

Figure 87 focuses on key caching. The tests confirm that calling `get_server_keys()` repeatedly returns the same active key rather than creating duplicates [Django, 2023].

```python
# --- Key Generation and Caching Tests ---
def test_generate_new_key(self):
    key = crypto_utils.generate_new_key()
    self.assertTrue(key.is_active)
    self.assertEqual(key.algorithm, "MLKEM1024")

def test_server_keys_existence(self):
    keys = crypto_utils.get_server_keys()
    self.assertIsNotNone(keys)

def test_server_keys_caching(self):
    keys1 = crypto_utils.get_server_keys()
    keys2 = crypto_utils.get_server_keys()
    self.assertEqual(keys1, keys2)
```

Figure 87: Server Key Caching Unit tests

Figure 88 demonstrates my message-level encryption tests. It checks that valid messages are decrypted properly, while tampered with ciphertext triggers decryption failures.

```python
# --- Message Encryption Tests ---
def test_encrypt_decrypt_message(self):
    message = "This is a secure message."
    encrypted_message = crypto_utils.encrypt_message(message)
    decrypted_message = crypto_utils.decrypt_message(encrypted_message)
    self.assertEqual(message, decrypted_message)

def test_message_invalid(self):
    message = "Another secure message"
    encrypted_message = crypto_utils.encrypt_message(message)
    # Tamper with the encrypted message.
    tampered = encrypted_message[:-5] + "12345"
    with self.assertRaises(Exception):
        crypto_utils.decrypt_message(tampered)
```

Figure 88: Message encryption/decryption and Tampering Unit tests

Figure 89 verifies my field-level encryption logic, ensuring that string fields (e.g., transaction details) can be securely stored and accurately recovered.

```python
# --- Field Encryption Tests ---
def test_encrypt_decrypt_field(self):
    value = "Test Field Value"
    salt = b'field_salt'
    info = b'field info'
    encrypted = crypto_utils.encrypt_field(value, salt, info)
    decrypted = crypto_utils.decrypt_field(encrypted, salt, info)
    self.assertEqual(value, decrypted)

def test_field_invalid_decryption(self):
    value = "Sensitive Data"
    salt = b'field_salt'
    info = b'field info'
    encrypted = crypto_utils.encrypt_field(value, salt, info)
    # Tamper with the encrypted data.
    tampered = encrypted[:-5] + "XXXXX"
    with self.assertRaises(Exception):
        crypto_utils.decrypt_field(tampered, salt, info)
```

Figure 89: Field Encryption Unit tests

Finally, **Figure 90** illustrates tests for encrypting and decrypting numeric balances, confirming that tampering with the encrypted data raises an exception and that valid data is accurately restored.

```python
# --- Balance Encryption Tests ---
def test_encrypt_decrypt_balance(self):
    balance = Decimal('12345.67')
    encrypted_balance = crypto_utils.encrypt_balance(balance)
    decrypted_balance = crypto_utils.decrypt_balance(encrypted_balance)
    self.assertEqual(balance, decrypted_balance)

def test_encrypt_balance_invalid(self):
    balance = Decimal('9999.99')
    encrypted_balance = crypto_utils.encrypt_balance(balance)
    # Tamper with encrypted data to force decryption failure.
    tampered = encrypted_balance[:-5] + "00000"
    with self.assertRaises(Exception):
        crypto_utils.decrypt_balance(tampered)
```

Figure 90: Balance Encryption Unit tests

Unit Testing Results

When running the tests as seen in **Figure 91**, all tests ran successfully except the *test_encrypt_balance_zero* test as shown in **Figure 92**.

```
python manage.py test --keepdb
```

Figure 91: Running Unit Tests (With –keepdb flag due to hosting constraints)

```
======================================================================
ERROR: test_encrypt_balance_zero (bank.test_crypto_utils.EncryptionTests.test_encrypt_balance_zero)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\stanl\Downloads\Bank\finance_app\bank\test_crypto_utils.py", line 124, in test_encrypt_balance_zero
    encrypted = crypto_utils.encrypt_balance(balance)
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\stanl\Downloads\Bank\finance_app\bank\crypto_utils.py", line 82, in encrypt_balance
    raise ValueError("Balance must be provided.")
ValueError: Balance must be provided.

----------------------------------------------------------------------
Ran 34 tests in 27.254s

FAILED (errors=1)
```

Figure 92: Unit Test Errors for Zero Balance Encryption

This error occurred because the encryption function initially used the condition shown in **Figure 93**, which rejected zero values due to Python treating zero as false. As a result, encrypting a valid zero balance raised a `ValueError`. To fix this, the condition was modified (see **Figure 94**) so that an error is raised only if no value is provided at all [Beizer, 1995].

```
if not balance:
    raise ValueError("Balance must be provided.")
```

Figure 93: Original Condition Rejecting Zero Balance

```
if balance is None:
    raise ValueError("Balance must be provided.")
```

Figure 94: Modified Condition Accepting Zero Balance

After fixing this issue, all unit tests passed (see **Figure 95**) confirming that all cryptographic functions—encryption, decryption, hashing, and key management—work correctly under normal and adversarial conditions [Myers et al., 2011].

```
(venv) C:\Users\stanl\Downloads\Bank\finance_app>python manage.py test --keepdb
Found 34 test(s).
Using existing test database for alias 'default'...
C:\Users\stanl\Downloads\Bank\venv\Lib\site-packages\django\db\backends\postgre
on database when it's not needed (for example, when running tests). Django was
  warnings.warn(
System check identified no issues (0 silenced).
..................................
----------------------------------------------------------------------
Ran 34 tests in 32.373s

OK
Preserving test database for alias 'default'...
```

Figure 95: Running Unit Tests (Successful)

5.2 Integration Testing

Integration tests were created to ensure that the different modules of the system work together seamlessly [Sommerville, 2015]. The following figures present screenshots of tests for key end-to-end scenarios, grouped by functionality, demonstrating both successful operations and proper error handling.

Admin Functionality Testing: **Figure 96** displays the test for the Admin Create User functionality, including validation error messages (e.g. for short passwords) [Django, 2023]. **Figure 97** shows the test verifying the Admin User Detail/Deletion process.

```python
def test_admin_create_user_view(self):
    self.client.login(username='admin', password='pass')
    url = reverse('admin_create_user')
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
    valid_post_data = {
        'username': 'newclient',
        'email': 'newclient@example.com',
        'password1': 'StrongPass123!',
        'password2': 'StrongPass123!',
        'role': 'client',
        'advisor': self.advisor_user.id,
    }
    response = self.client.post(url, valid_post_data, follow=True)
    new_user = User.objects.filter(username='newclient').first()
    self.assertIsNotNone(new_user, "New user should be created for valid data.")
    self.assertEqual(new_user.profile.role, 'client')
    self.assertEqual(new_user.profile.advisor, self.advisor_user)
    invalid_post_data = {
        'username': 'failclient',
        'email': 'fail@example.com',
        'password1': 'Short',
        'password2': 'Short',
        'role': 'client',
        'advisor': self.advisor_user.id,
    }
    response = self.client.post(url, invalid_post_data, follow=True)
    content = response.content.decode().lower()
    self.assertIn("too short", content,
                  "Expected error message for short password not found.")
    self.client.logout()
    self.client.login(username='advisor', password='pass')
    response = self.client.get(url, follow=True)
    content = response.content.decode().lower()
    self.assertIn("access denied", content,
                  "Advisor should see access-denied message when accessing admin create user view.")
```

Figure 96: Test for Admin Create User functionality

```python
def test_admin_user_detail_view(self):
    self.client.login(username='admin', password='pass')
    url = reverse('admin_user_detail', args=[self.client_user.id])
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
    self.assertContains(response, self.client_user.username)

    self.client.logout()
    self.client.login(username='advisor', password='pass')
    response = self.client.get(url, follow=True)
    content = response.content.decode().lower()
    self.assertIn("access denied", content,
                  "Advisor should see access-denied message when accessing admin user detail view.")

def test_admin_user_delete_view(self):
    self.client.login(username='admin', password='pass')
    user_count_before = User.objects.count()
    url = reverse('admin_user_delete', args=[self.client_user.id])
    # Check delete confirmation page contains expected text.
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
    self.assertIn("delete", response.content.decode().lower())
    # Perform deletion and verify state change.
    response = self.client.post(url, follow=True)
    user_count_after = User.objects.count()
    self.assertEqual(user_count_after, user_count_before - 1,
                     "User count should decrease by one after deletion.")
```

Figure 97: Test for Admin User Detail/Deletion functionality

Advisor Transaction Testing: **Figure 98** illustrates the test for an attempt to buy with insufficient funds, and **Figure 99** displays the test for a valid advisor buy transaction [Django, 2023]. **Figure 100** shows the tests for valid sell transactions. **Figures 101 and 102** present the tests for verifying that negative share inputs are rejected and for attempts to sell when no holdings exist, respectively. Finally, **Figure 103** captures the test for the combined advisor transaction and normal view scenario.

```python
@patch('bank.views.yf.Ticker')
def test_advisor_transaction_buy_insufficient_funds(self, mock_ticker):
    fake_df = pd.DataFrame({'Close': [150.00]})
    mock_instance = MagicMock()
    mock_instance.history.return_value = fake_df
    mock_ticker.return_value = mock_instance

    self.client.login(username='advisor', password='pass')
    url = reverse('advisor_transaction')
    # Set insufficient funds.
    self.client_user.profile.balance = Decimal('100')
    self.client_user.profile.save()
    post_data = {
        'client': self.client_user.id,
        'ticker': 'AAPL',
        'shares': '10',
        'action': 'buy',
    }
    response = self.client.post(url, post_data, follow=True)
    # Verify that no holding was created and balance remains unchanged.
    holding = Holding.objects.filter(portfolio=self.client_user.portfolio, stock=self.stock).first()
    self.assertIsNone(holding, "No holding should be created when funds are insufficient.")
    self.client_user.profile.refresh_from_db()
    self.assertEqual(self.client_user.profile.balance, Decimal('100'))
```

Figure 98: Test for advisor buy transaction with insufficient funds

```python
@patch('bank.views.yf.Ticker')
def test_advisor_transaction_buy_valid(self, mock_ticker):
    # Patch yfinance.Ticker to return a fake history DataFrame with Close=150.00.
    fake_df = pd.DataFrame({'Close': [150.00]})
    mock_instance = MagicMock()
    mock_instance.history.return_value = fake_df
    mock_ticker.return_value = mock_instance

    self.client.login(username='advisor', password='pass')
    url = reverse('advisor_transaction')
    # Ensure client has sufficient funds.
    self.client_user.profile.balance = Decimal('10000')
    self.client_user.profile.save()
    post_data = {
        'client': self.client_user.id,
        'ticker': 'AAPL',
        'shares': '10',
        'action': 'buy',
    }
    response = self.client.post(url, post_data, follow=True)
    # Verify that a holding was created.
    holding = Holding.objects.filter(portfolio=self.client_user.portfolio, stock=self.stock).first()
    self.assertIsNotNone(holding, "Holding should be created for a valid buy transaction.")
    self.assertEqual(holding.shares, Decimal('10'))
    # Verify that the client's balance decreased correctly (10 * 150 = 1500).
    expected_balance = Decimal('10000') - (Decimal('10') * Decimal('150.00'))
    self.client_user.profile.refresh_from_db()
    self.assertEqual(self.client_user.profile.balance, expected_balance)
    # Verify that an InvestmentTransaction record was created.
    tx = InvestmentTransaction.objects.filter(portfolio=self.client_user.portfolio).first()
    self.assertIsNotNone(tx)
```

Figure 99: Test for valid advisor buy transaction

```python
@patch('bank.views.yf.Ticker')
def test_advisor_transaction_sell_valid(self, mock_ticker):
    fake_df = pd.DataFrame({'Close': [150.00]})
    mock_instance = MagicMock()
    mock_instance.history.return_value = fake_df
    mock_ticker.return_value = mock_instance
    self.client.login(username='advisor', password='pass')
    url = reverse('advisor_transaction')
    # Pre-create a holding with 10 shares.
    Holding.objects.create(
        portfolio=self.client_user.portfolio,
        stock=self.stock,
        shares=Decimal('10')
    )
    self.client_user.profile.balance = Decimal('5000')
    self.client_user.profile.save()
    post_data = {
        'client': self.client_user.id,
        'ticker': 'AAPL',
        'shares': '10',
        'action': 'sell',
    }
    response = self.client.post(url, post_data, follow=True)
    # Verify that the holding is removed.
    holding = Holding.objects.filter(portfolio=self.client_user.portfolio, stock=self.stock).first()
    self.assertIsNone(holding, "Holding should be removed after selling all shares.")
    # Verify that the client's balance increased correctly (10 * 150 = 1500).
    expected_balance = Decimal('5000') + (Decimal('10') * Decimal('150.00'))
    self.client_user.profile.refresh_from_db()
    self.assertEqual(self.client_user.profile.balance, expected_balance)
    tx = InvestmentTransaction.objects.filter(portfolio=self.client_user.portfolio).first()
    self.assertIsNotNone(tx, "InvestmentTransaction record should be created for valid sell transaction.")
```

Figure 100: Test for valid advisor sell transaction

```python
@patch('bank.views.yf.Ticker')
def test_advisor_transaction_buy_negative_shares(self, mock_ticker):
    fake_df = pd.DataFrame({'Close': [150.00]})
    mock_instance = MagicMock()
    mock_instance.history.return_value = fake_df
    mock_ticker.return_value = mock_instance

    initial_balance = self.client_user.profile.balance
    initial_holdings_count = Holding.objects.filter(
        portfolio=self.client_user.portfolio, stock=self.stock
    ).count()
    initial_tx_count = InvestmentTransaction.objects.filter(
        portfolio=self.client_user.portfolio
    ).count()
    self.client.login(username='advisor', password='pass')
    url = reverse('advisor_transaction')
    post_data = {
        'client': self.client_user.id,
        'ticker': 'AAPL',
        'shares': '-10',  # Negative share input.
        'action': 'buy',
    }
    response = self.client.post(url, post_data, follow=True)
    # Retrieve messages from the response.
    messages_list = list(get_messages(response.wsgi_request))
    error_found = any("must buy at least" in message.message.lower() for message in messages_list)
    self.assertTrue(error_found, "Expected error message for negative share input not found.")
    # Verify that no new holding was created or modified.
    holdings_count = Holding.objects.filter(
        portfolio=self.client_user.portfolio, stock=self.stock
    ).count()
    self.assertEqual(holdings_count, initial_holdings_count,
                     "Holding record should not be created/modified for negative share input.")
    # Verify that no new InvestmentTransaction record was created.
    tx_count = InvestmentTransaction.objects.filter(
        portfolio=self.client_user.portfolio
    ).count()
    self.assertEqual(tx_count, initial_tx_count,
                     "InvestmentTransaction record should not be created for negative share input.")
    # Verify that the client's balance remains unchanged.
    self.client_user.profile.refresh_from_db()
    self.assertEqual(self.client_user.profile.balance, initial_balance,
                     "Client's balance should not change for negative share input.")
```

Figure 101: Test for rejection of negative share input in advisor buy transaction

```python
def test_advisor_transaction_sell_insufficient_shares(self):
    self.client.login(username='advisor', password='pass')
    url = reverse('advisor_transaction')
    holding = Holding.objects.create(
        portfolio=self.client_user.portfolio,
        stock=self.stock,
        shares=Decimal('5')
    )
    post_data = {
        'client': self.client_user.id,
        'ticker': 'AAPL',
        'shares': '10',  # Trying to sell more than held.
        'action': 'sell',
    }
    response = self.client.post(url, post_data, follow=True)
    # Verify that the holding remains unchanged.
    holding.refresh_from_db()
    self.assertEqual(holding.shares, Decimal('5'),
                     "Holding should remain unchanged when attempting to sell insufficient shares.")
    self.client_user.profile.refresh_from_db()
    self.assertEqual(self.client_user.profile.balance, Decimal('10000'),
                     "Client's balance should not change when selling insufficient shares.")
```

Figure 102: Test for advisor sell transaction with no holdings

```python
def test_advisor_view_access(self):
    self.client.login(username='advisor', password='pass')
    url = reverse('advisor')
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
    self.assertIn("advisor", response.content.decode().lower())
    self.client.logout()
    self.client.login(username='client', password='pass')
    response = self.client.get(url, follow=True)
    self.assertIn("portfolio", response.content.decode().lower(),
                  "Client should be redirected to portfolio when accessing advisor view.")

@patch('bank.views.yf.Ticker')
def test_advisor_transaction_buy_valid(self, mock_ticker):
    # Patch yfinance.Ticker to return a fake history DataFrame with Close=150.00.
    fake_df = pd.DataFrame({'Close': [150.00]})
    mock_instance = MagicMock()
    mock_instance.history.return_value = fake_df
    mock_ticker.return_value = mock_instance

    self.client.login(username='advisor', password='pass')
    url = reverse('advisor_transaction')
    # Ensure client has sufficient funds.
    self.client_user.profile.balance = Decimal('10000')
    self.client_user.profile.save()
    post_data = {
        'client': self.client_user.id,
        'ticker': 'AAPL',
        'shares': '10',
        'action': 'buy',
    }
    response = self.client.post(url, post_data, follow=True)
    # Verify that a holding was created.
    holding = Holding.objects.filter(portfolio=self.client_user.portfolio, stock=self.stock).first()
    self.assertIsNotNone(holding, "Holding should be created for a valid buy transaction.")
    self.assertEqual(holding.shares, Decimal('10'))
    # Verify that the client's balance decreased correctly (10 * 150 = 1500).
    expected_balance = Decimal('10000') - (Decimal('10') * Decimal('150.00'))
    self.client_user.profile.refresh_from_db()
    self.assertEqual(self.client_user.profile.balance, expected_balance)
    # Verify that an InvestmentTransaction record was created.
    tx = InvestmentTransaction.objects.filter(portfolio=self.client_user.portfolio).first()
    self.assertIsNotNone(tx)
```

Figure 103: Test for combined advisor transaction and normal view scenario

Advisor Client Detail Testing: **Figure 104** displays the test for the Advisor Client Detail functionality, confirming that client-specific portfolio information is correctly presented [Django, 2023].

```python
def test_advisor_view_access(self):
    self.client.login(username='advisor', password='pass')
    url = reverse('advisor')
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
    self.assertIn("advisor", response.content.decode().lower())
    self.client.logout()
    self.client.login(username='client', password='pass')
    response = self.client.get(url, follow=True)
    self.assertIn("portfolio", response.content.decode().lower(),
                  "Client should be redirected to portfolio when accessing advisor view.")
```

Figure 104: Test for Advisor Client Detail functionality

Non-Advisor Access Testing: **Figure 105** displays the integration test confirming that non-advisor users are correctly redirected when attempting to access advisor-only views [Django, 2023].

```
def test_non_advisor_access_to_advisor_views(self):
    advisor_url = reverse('advisor')
    self.client.login(username='client', password='pass')
    response = self.client.get(advisor_url, follow=True)
    self.assertIn("portfolio", response.content.decode().lower(),
                  "Client should be redirected to portfolio when accessing advisor view.")
    trans_url = reverse('advisor_transaction')
    response = self.client.get(trans_url, follow=True)
    self.assertIn("portfolio", response.content.decode().lower(),
                  "Client should be redirected to portfolio when accessing advisor transaction view.")
```

Figure 105: Test for Non-Advisor Access to Advisor Views

Advisor Messaging Testing: **Figure 106** shows the integration test outputs for sending messages as an advisor, ensuring valid messages are delivered and invalid inputs (e.g., missing recipient or message text) are handled correctly [Django, 2023].

```
@patch('django.template.loader.get_template')
def test_advisor_message_view_invalid(self, mock_get_template):
    dummy_template = type("DummyTemplate", (), {"render": lambda self, context, request=None: "dummy"})()
    mock_get_template.return_value = dummy_template

    self.client.login(username='advisor', password='pass')
    url = reverse('advisor_message')
    initial_message_count = Message.objects.count()
    post_data = {'recipient': '', 'message': 'Advice'}
    response = self.client.post(url, post_data, follow=True)
    self.assertEqual(Message.objects.count(), initial_message_count,
                     "No message should be created if recipient is missing.")
    post_data = {'recipient': self.client_user.username, 'message': ''}
    response = self.client.post(url, post_data, follow=True)
    self.assertEqual(Message.objects.count(), initial_message_count,
                     "No message should be created if message text is missing.")
    post_data = {'recipient': 'nonexistent', 'message': 'Advice'}
    response = self.client.post(url, post_data, follow=True)
    self.assertEqual(Message.objects.count(), initial_message_count,
                     "No message should be created if recipient does not exist.")

@patch('django.template.loader.get_template')
def test_advisor_message_view_valid(self, mock_get_template):
    dummy_template = type("DummyTemplate", (), {"render": lambda self, context, request=None: "dummy"})()
    mock_get_template.return_value = dummy_template

    self.client.login(username='advisor', password='pass')
    url = reverse('advisor_message')
    initial_message_count = Message.objects.count()
    post_data = {'recipient': self.client_user.username, 'message': 'Investment advice'}
    response = self.client.post(url, post_data, follow=True)
    self.assertEqual(Message.objects.count(), initial_message_count + 1,
                     "A valid message should be created when recipient and message text are provided.")
    msg = Message.objects.filter(sender=self.advisor_user, recipient=self.client_user).first()
    self.assertIsNotNone(msg, "Message record should exist for a valid message.")
```

Figure 106: Test for Advisor Messaging Functionality

Client Detail Testing: **Figure 107** displays the integration test verifying that client detail views are properly shown to authorised users and restricted for others [Django, 2023].

```
def test_advisor_client_detail_view(self):
    self.client.login(username='advisor', password='pass')
    url = reverse('advisor_client_detail', args=[self.client_user.id])
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
    self.assertIn(self.client_user.username, response.content.decode())
    self.client.logout()
    self.client.login(username='client', password='pass')
    response = self.client.get(url, follow=True)
    self.assertIn("portfolio", response.content.decode().lower(),
                  "Client accessing advisor client detail should be redirected to portfolio.")
```

Figure 107: Test for Client Detail View Functionality

Integration Testing Results

All but one integration test passed as seen in **Figure 108**. This failed because I was able to purchase a negative amount of stock as there were no bounds checks in place for the advisor transaction view. In order to fix this, I simply added bounds checking that ensured that the amount of stock purchased had to be at least *0.01* share [Beizer, 1995] (see **Figure 109**).

```
======================================================
FAIL: test_advisor_transaction_buy_negative_shares
------------------------------------------------------
Traceback (most recent call last):
  File "C:\Program Files\WindowsApps\PythonSoftware
    return func(*newargs, **newkeywargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\stanl\Downloads\Bank\finance_app\b
    self.assertTrue(
AssertionError: False is not true : Expected error

------------------------------------------------------
```

Figure 108: Integration Error in Admin Functionality

```
if action == 'buy':
    if shares <= 0:
        messages.error(request, "Must buy at least 0.01 share.")
```

Figure 109: Transaction Handling Bounds Checking Fix

This testing process not only validated the functionality of individual functions, but also ensured that all components of the system integrate seamlessly, resulting in a secure and robust financial application [Sommerville, 2015].

6 Conclusion

This report details a cryptosystem for *MyFinance Inc.* that leverages robust encryption techniques including AES-256 with GCM for authenticated encryption, HKDF for secure key derivation, and MLKEM-1024 for quantum-resistant key management and rotation. The system secures sensitive financial data by ensuring confidentiality, integrity, and authenticity while facilitating secure communication between the company and its clients. It also integrates role-based access control into Django's authentication framework for clients, financial advisors, and system administrators, and employs HMAC to maintain transaction data integrity. In addition, the system features scheduled background tasks that update stock prices every 60 seconds independently of user activity. Despite challenges related to secure key management and scalability, thorough testing confirmed that encryption, decryption, authentication, and user workflows operate reliably. Future enhancements could include multi-factor authentication, performance optimisations for background tasks, and further hardening of the key management system as new quantum-resistant algorithms emerge.

References

Mattias Aabmets. Introduction to quantcrypt: A python library for quantum-resistant cryptography. *Medium*, 6 February, 2024. Available at: `https://medium.com/@mattias.aabmets/introduction-to-quantcrypt-a-python-library-for-quantum-resistant-cryptography-00faec1cc032` [Accessed 29 March 2025].

Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, New York, 3rd edition, 2020.

Ran Aroussi. yfinance documentation, 2025. Available at: `https://ranaroussi.github.io/yfinance/` [Accessed 29 March 2025].

Elaine Barker. Recommendation for key management: Part 1 – general (nist sp 800-57 part 1 rev. 5). Technical report, National Institute of Standards and Technology, Gaithersburg, MD, 2020.

Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1995.

Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302, Saarbrücken, Germany, March 2016.

Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. 2020. [Online]. Available at: `https://crypto.stanford.edu/~dabo/cryptobook/` [Accessed 29 March 2025].

Joseph Bonneau, Cormac Herley, Paul C. Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 553–567, 2012.

Rainer Böhme. A roadmap toward a theory of intrusion detection. In *Proceedings of the 19th International Conference on World Wide Web*, pages 1063–1072, 2012.

Lily Chen, Meltem Chen, and Scott Jordan. Report on post-quantum cryptography, 2016. Available at: `https://csrc.nist.gov/projects/post-quantum-cryptography` [Accessed 29 March 2025].

Django. Django documentation (version 4.2), 2023. Available at: `https://docs.djangoproject.com/en/4.2/` [Accessed 29 March 2025].

Morris Dworkin. Nist special publication 800-38d: Recommendation for block cipher modes of operation – galois/counter mode (gcm) and gmac. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, 2007.

European Union. General data protection regulation (gdpr), 2018.

David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, 2003.

Hugo Krawczyk and Pasi Eronen. Hmac-based extract-and-expand key derivation function (hkdf). Technical Report RFC 5869, Internet Engineering Task Force (IETF), 2010. Available at: `https://datatracker.ietf.org/doc/html/rfc5869` [Accessed 29 March 2025].

Alfred Menezes and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.

Michele Mosca. Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security & Privacy*, 16(5):38–41, 2018.

Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley, New York, 3rd edition, 2011.

NIST. Fips pub 197: Advanced encryption standard (aes). Technical report, U.S. Department of Commerce, Washington, DC, 2001.

NIST. Fips pub 198-1: The keyed-hash message authentication code (hmac). Technical report, NIST, Gaithersburg, MD, 2008.

NIST. Fips pub 180-4: Secure hash standard (shs). Technical report, NIST, Gaithersburg, MD, 2015.

NIST. Fips pub 203: Module-lattice-based key-encapsulation mechanism (ml-kem) standard. Technical report, NIST, Gaithersburg, MD, 2024.

OWASP. Logging cheat sheet, 2025a. *OWASP Cheat Sheet Series*. Available at: `https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html` [Accessed 29 March 2025].

OWASP. Manipulator-in-the-middle attack, 2025b. *OWASP Web Security Knowledge Base*. Available at: `https://owasp.org/www-community/attacks/Manipulator-in-the-middle_attack` [Accessed 29 March 2025].

OWASP. Password storage cheat sheet, 2025c. *OWASP Cheat Sheet Series*. Available at: `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html` [Accessed 29 March 2025].

PyCA. Cryptography library documentation, 2023. Available at: `https://cryptography.io/en/latest/` [Accessed 29 March 2025].

Eric Rescorla. The transport layer security (tls) protocol version 1.3. Technical Report RFC 8446, Internet Engineering Task Force (IETF), 2018. Available at: `https://datatracker.ietf.org/doc/html/rfc8446` [Accessed 29 March 2025].

Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

Karen A. Scarfone and Murugiah Souppaya. Guide to computer security log management (nist sp 800-92). Technical report, National Institute of Standards and Technology, Gaithersburg, MD, 2006.

Ian Sommerville. *Software Engineering*. Pearson, Boston, 10th edition, 2015.

William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson, Upper Saddle River, NJ, 7th edition, 2017.

Verizon. 2021 data breach investigations report, 2021. Available at: `https://www.verizon.com/business/resources/reports/dbir/` [Accessed 29 March 2025].