Here is a detailed, step-by-step breakdown for an AI agent to implement a large file upload system from scratch using React, Django, and Azure Data Lake Storage (ADLS).

The strategy is to use the **chunking (block-based) approach**. The React client will slice the file and send chunks to the Django server. The Django server will act as a secure proxy, receiving these chunks and forwarding them to ADLS without storing them locally.

Part 1: Backend Implementation (Django)

Step 1: Set Up the Django Project

1. **Create a virtual environment and install packages:**
   Bash
   ```
   python -m venv venv
   # On macOS/Linux
   source venv/bin/activate
   # On Windows
   # venv\Scripts\activate
   pip install django djangorestframework django-cors-headers "azure-storage-blob>=12.13.0" python-dotenv
   ```

2. **Create the Django project and app:**
   Bash
   ```
   django-admin startproject adls_uploader
   cd adls_uploader
   python manage.py startapp uploader
   ```

Step 2: Configure the Django Project

1. **Update adls_uploader/settings.py:**
   - Add the new app and necessary frameworks to INSTALLED_APPS.
   - Add CorsMiddleware for cross-origin requests from React.

   Python
   ```
   # adls_uploader/settings.py
   INSTALLED_APPS = [
       # ...
       'rest_framework',
       'uploader',
       'corsheaders',
   ]

   MIDDLEWARE = [
       # ...
   ```

```
    'corsheaders.middleware.CorsMiddleware', # Add this before CommonMiddleware
    'django.middleware.common.CommonMiddleware',
    # ...
]

# Allow your React app to connect
CORS_ALLOWED_ORIGINS = [
    "http://localhost:3000",
]
```

2. **Add Azure Credentials:**
    - Create a .env file in the project root (adls_uploader/).
    - **Do not commit this file to version control.**

```
Ini, TOML
# .env
AZURE_STORAGE_CONNECTION_STRING="your_full_connection_string_from_azure_portal"
AZURE_CONTAINER_NAME="your_container_name"
```

    - Load these variables in settings.py:

```
Python
# adls_uploader/settings.py at the top
import os
from dotenv import load_dotenv
load_dotenv()
```

Step 3: Create the API Endpoints

1. **Define the views in uploader/views.py:**
    - **UploadChunkView**: Receives a single file chunk and uploads it to Azure as an uncommitted block.
    - **CommitUploadView**: Receives the list of block IDs and commits them to form the final file.

```
Python
# uploader/views.py
import os
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from azure.storage.blob import BlobServiceClient, BlobBlock

# Initialize the Blob Service Client from the connection string
connection_string = os.getenv("AZURE_STORAGE_CONNECTION_STRING")
container_name = os.getenv("AZURE_CONTAINER_NAME")
blob_service_client = BlobServiceClient.from_connection_string(connection_string)
```

```python
class UploadChunkView(APIView):
    def post(self, request, *args, **kwargs):
        file_chunk = request.data['chunk']
        block_id = request.data['block_id']
        file_name = request.data['file_name']

        try:
            blob_client = blob_service_client.get_blob_client(container=container_name, blob=file_name)
            blob_client.stage_block(block_id=block_id, data=file_chunk)
            return Response({"message": f"Block {block_id} uploaded successfully"}, status=status.HTTP_201_CREATED)
        except Exception as e:
            return Response({"error": str(e)}, status=status.HTTP_500_INTERNAL_SERVER_ERROR)

class CommitUploadView(APIView):
    def post(self, request, *args, **kwargs):
        file_name = request.data['file_name']
        block_ids = request.data['block_ids']

        try:
            blob_client = blob_service_client.get_blob_client(container=container_name, blob=file_name)
            block_list = [BlobBlock(block_id=b_id) for b_id in block_ids]
            blob_client.commit_block_list(block_list)

            final_url = blob_client.url
            return Response({"message": "File uploaded successfully", "url": final_url}, status=status.HTTP_200_OK)
        except Exception as e:
            return Response({"error": str(e)}, status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

Step 4: Wire Up the URLs

1. **Create uploader/urls.py:**

   ```python
   # uploader/urls.py
   from django.urls import path
   from .views import UploadChunkView, CommitUploadView

   urlpatterns = [
       path('upload-chunk/', UploadChunkView.as_view(), name='upload-chunk'),
       path('commit-upload/', CommitUploadView.as_view(), name='commit-upload'),
   ]
   ```

2. **Include the app URLs in adls_uploader/urls.py:**

   ```python
   ```

```
# adls_uploader/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('uploader.urls')),
]
```

Step 5: Run the Backend Server

Bash

```
python manage.py migrate
python manage.py runserver
```

The Django API is now running at http://localhost:8000.

Part 2: Frontend Implementation (React)

Step 1: Set Up the React Project

1. **Create a new React app and install dependencies:**
   Bash
   ```
   npx create-react-app adls-client
   cd adls-client
   npm install axios react-dropzone
   ```

Step 2: Create the File Upload Component

1. **Create a new component file src/FileUpload.js:** This component will handle the entire client-side logic.
   JavaScript
   ```
   // src/FileUpload.js
   import React, { useCallback, useState } from 'react';
   import { useDropzone } from 'react-dropzone';
   import axios from 'axios';

   const CHUNK_SIZE = 10 * 1024 * 1024; // 10MB Chunks
   const DJANGO_API_URL = 'http://localhost:8000/api';

   const FileUpload = () => {
   ```

```
const [uploadProgress, setUploadProgress] = useState(0);
const [fileUrl, setFileUrl] = useState(null);
const [error, setError] = useState(null);

const onDrop = useCallback(async (acceptedFiles) => {
  const file = acceptedFiles[0];
  if (!file) return;

  // Reset state for new upload
  setUploadProgress(0);
  setFileUrl(null);
  setError(null);

  const totalChunks = Math.ceil(file.size / CHUNK_SIZE);
  const blockIds = [];
  const uploadPromises = [];

  for (let i = 0; i < totalChunks; i++) {
    const start = i * CHUNK_SIZE;
    const end = Math.min(start + CHUNK_SIZE, file.size);
    const chunk = file.slice(start, end);

    // Block ID must be a Base64 encoded string.
    // The string itself can be anything, but it must be the same length for all blocks.
    // Padding with leading zeros ensures consistent length.
    const blockId = btoa(String(i).padStart(5, '0'));
    blockIds.push(blockId);

    const formData = new FormData();
    formData.append('chunk', chunk);
    formData.append('block_id', blockId);
    formData.append('file_name', file.name);

    const promise = axios.post(`${DJANGO_API_URL}/upload-chunk/`, formData)
      .then(() => {
        // Update progress after each successful chunk upload
        setUploadProgress(prev => prev + (1 / totalChunks) * 100);
      });

    uploadPromises.push(promise);
  }

  try {
```

```
    // Wait for all chunk uploads to complete
    await Promise.all(uploadPromises);

    // All chunks are staged, now commit them
    const commitResponse = await axios.post(`${DJANGO_API_URL}/commit-upload/`, {
      file_name: file.name,
      block_ids: blockIds,
    });

    setFileUrl(commitResponse.data.url);

  } catch (err) {
    console.error("Upload failed:", err);
    setError("Upload failed. Please try again.");
    setUploadProgress(0); // Reset progress on failure
  }
 }, []);

 const { getRootProps, getInputProps, isDragActive } = useDropzone({ onDrop, multiple:
false });

 return (
  <div style={styles.container}>
   <div {...getRootProps()} style={isDragActive ? styles.dropzoneActive : styles.dropzone}>
    <input {...getInputProps()} />
    <p>Drag 'n' drop a large file here, or click to select a file</p>
   </div>
   {uploadProgress > 0 && (
    <div style={styles.progressContainer}>
     <p>Uploading... {Math.round(uploadProgress)}%</p>
     <div style={styles.progressBarBackground}>
      <div style={{...styles.progressBarFill, width: `${uploadProgress}%`}}></div>
     </div>
    </div>
   )}
   {fileUrl && (
    <div style={styles.successMessage}>
     <p>✅ File uploaded successfully!</p>
     <a href={fileUrl} target="_blank" rel="noopener noreferrer">View File</a>
    </div>
   )}
   {error && <p style={styles.errorMessage}>{error}</p>}
  </div>
 );
};
```

```
// Basic styling for the component
const styles = {
  container: { width: '60%', margin: '50px auto', textAlign: 'center' },
  dropzone: { border: '2px dashed #cccccc', borderRadius: '10px', padding: '20px', cursor: 'pointer' },
  dropzoneActive: { border: '2px dashed #007bff' },
  progressContainer: { marginTop: '20px' },
  progressBarBackground: { height: '20px', width: '100%', backgroundColor: '#e0e0e0', borderRadius: '10px' },
  progressBarFill: { height: '100%', backgroundColor: '#4caf50', borderRadius: '10px' },
  successMessage: { color: 'green', marginTop: '20px' },
  errorMessage: { color: 'red', marginTop: '20px' },
};

export default FileUpload;
```

Step 3: Use the Component in the Main App

1. **Replace the content of src/App.js:**

```
JavaScript
// src/App.js
import React from 'react';
import FileUpload from './FileUpload';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>Large File Uploader to Azure ADLS</h1>
        <FileUpload />
      </header>
    </div>
  );
}

export default App;
```

Step 4: Run the Frontend App

```
Bash

npm start
```

The React app will open at http://localhost:3000. You can now drag and drop a large file to begin the chunked upload process.