Abacus.AI REST API Integration Research

Overview

This document contains comprehensive research on integrating with the Abacus.AI REST API, specifically focusing on pipeline execution results, authentication mechanisms, response structures, and best practices.

---

1. API Endpoints for Fetching Pipeline Execution Results and Data

Pipeline Management Endpoints

Abacus.AI provides a comprehensive set of endpoints for managing and querying pipeline execution results:

Core Pipeline Endpoints

Pipeline Creation & Management:
- `POST /api/v0/createPipeline` - Creates a new pipeline for automated workflows
- `GET /api/v0/describePipeline` - Retrieves detailed information about a specific pipeline using its ID
- `GET /api/v0/listPipelines` - Lists all pipelines in a project
- `PATCH /api/v0/updatePipeline` - Updates pipeline properties
- `DELETE /api/v0/deletePipeline` - Deletes a pipeline

Pipeline Execution:
- `POST /api/v0/runPipeline` - Triggers execution of a specified pipeline version
- `GET /api/v0/describePipelineVersion` - Fetches details about a particular pipeline version, including execution status and results
- `GET /api/v0/listPipelineVersions` - Lists all versions of a pipeline

Pipeline Results & Logs:
- `GET /api/v0/listPipelineVersionLogs` - Retrieves logs from a pipeline version execution
- `GET /api/v0/getStepVersionLogs` - Gets detailed logs from specific pipeline steps
- `GET /api/v0/describePipelineVersion` - Returns metrics, logs, and execution outcomes for a pipeline version

Pipeline Scheduling:
- `POST /api/v0/pausePipelineRefreshSchedule` - Pauses scheduled pipeline executions
- `POST /api/v0/resumePipelineRefreshSchedule` - Resumes scheduled pipeline executions

Refresh Pipeline Runs

For tracking overall pipeline execution status:
- **RefreshPipelineRun** objects track the overall status of a refresh that can span multiple resources (dataset versions, model training, etc.)

Feature Group & Data Retrieval Endpoints

Since pipelines often process data through feature groups:

Feature Group Operations:
- `GET /api/v0/describeFeatureGroup` - Retrieves feature group details
- `GET /api/v0/describeFeatureGroupVersion` - Gets specific feature group version data
- `POST /api/v0/executeFeatureGroupOperation` - Executes SQL queries on feature groups
- `GET /api/v0/exportFeatureGroupVersionToFileConnector` - Exports feature group results

Dataset Operations:
- `GET /api/v0/describeDataset` - Retrieves dataset information
- `GET /api/v0/describeDatasetVersion` - Gets specific dataset version details
- `GET /api/v0/getDatasetSchema` - Retrieves dataset schema information

Prediction & Model Results

For pipelines involving model predictions:
- `POST /api/v0/predict` - Makes predictions using deployed models
- `POST /api/v0/predictMultiple` - Batch prediction requests
- `GET /api/v0/describeBatchPrediction` - Retrieves batch prediction results
- `GET /api/v0/describeBatchPredictionVersion` - Gets specific batch prediction version results

Base API URL

All endpoints use the base URL:

```
https://api.abacus.ai
```

Complete endpoint format:

```
https://api.abacus.ai/api/v0/{method_name}
```

---

2. Authentication Mechanism (API Key Headers and Format)

API Key Generation

1. Navigate to the **API Keys Dashboard** in Abacus.AI
2. Click **“Generate new API Key”**
3. Optionally add a custom tag for identification
4. Securely store the generated key (never commit to public repositories)

API Key Management

API keys in the dashboard display:
- **CREATED AT**: Timestamp of key creation (PST)
- **API KEY**: The actual key value
- **TAG**: Custom identifier for the key
- **ACTIONS**: Copy/delete operations

Authentication Header Format

Every API call requires an API key in the HTTP request header:

Header Name: `apiKey`

Header Format:

```
apiKey: YOUR_API_KEY
```

Authentication Examples

GET Request with Authentication

```
curl -H "apiKey: YOUR_API_KEY" "https://api.abacus.ai/api/v0/listProjects"
```

Example with actual key pattern:

```
curl -H "apiKey: 3210987**************86507975azz" \
     "https://api.abacus.ai/api/v0/listProjects"
```

POST Request with Authentication

```
curl -X POST \
     -H "Content-Type: application/json" \
     -H "apiKey: YOUR_API_KEY" \
     "https://api.abacus.ai/api/v0/createProject" \
     -d '{"name": "Customer Analytics", "useCase": "PREDICTIVE_MODELING"}'
```

Alternative Authentication Methods

For certain deployment-specific operations:
- **Deployment Token**: Can be used instead of API key for deployment-related endpoints
- Authentication format: Deployment ID + Deployment Token combination
- Useful for: `getApiEndpoint`, `startDeployment`, `stopDeployment`, etc.

Deployment Token Endpoints:
- `POST /api/v0/createDeploymentToken` - Creates deployment-specific tokens
- `GET /api/v0/listDeploymentTokens` - Lists all deployment tokens
- `DELETE /api/v0/deleteDeploymentToken` - Deletes a deployment token

Security Best Practices

1. **Never expose API keys in public repositories**
2. **Rotate keys periodically** - Delete old keys and generate new ones
3. **Use deployment tokens** for deployment-specific operations to limit scope
4. **Store keys securely** in environment variables or secure vaults
5. **Permanently revoke compromised keys** using the delete action

---

3. Response Data Structure from Pipelines

Standard Response Format

All Abacus.AI API responses follow a consistent structure:

Successful Response

```
{
  "success": true,
  "result": {
    // Response data specific to the endpoint
  }
}
```

Error Response

```
{
  "success": false,
  "error": "Error description message",
  "errorType": "ErrorTypeName"
}
```

Pipeline-Specific Response Structures

Pipeline Description Response

When calling `describePipeline`:

```
{
  "success": true,
  "result": {
    "pipelineId": "abc123def456",
    "name": "Data Processing Pipeline",
    "projectId": "proj_789xyz",
    "createdAt": "2024-01-15T10:30:00+00:00",
    "pipelineVersion": {
      "pipelineVersionId": "version_123",
      "status": "COMPLETE",
      "completedAt": "2024-01-15T11:45:00+00:00"
    },
    "steps": [
      // Array of pipeline steps
    ]
  }
}
```

Pipeline Version Response

When calling `describePipelineVersion`:

```
{
  "success": true,
  "result": {
    "pipelineVersionId": "version_123abc",
    "pipelineId": "pipeline_456def",
    "status": "COMPLETE",
    "createdAt": "2024-01-15T10:00:00+00:00",
    "completedAt": "2024-01-15T11:30:00+00:00",
    "steps": [
      {
        "stepId": "step_001",
        "stepName": "Data Ingestion",
        "status": "COMPLETE",
        "logs": "...",
        "metrics": {}
      }
    ],
    "error": null
  }
}
```


Pipeline Execution Logs

When calling `listPipelineVersionLogs` or `getStepVersionLogs`:


```
{
  "success": true,
  "result": {
    "logs":
"Detailed execution logs...\n[INFO] Processing started\n[INFO] Data loaded
successfully\n[INFO] Processing complete",
    "pipelineVersionId": "version_123",
    "timestamp": "2024-01-15T11:30:00+00:00"
  }
}
```


Run Pipeline Response

When calling `runPipeline`:


```
{
  "success": true,
  "result": {
    "pipelineVersionId": "version_789xyz",
    "status": "PENDING",
    "startedAt": "2024-01-15T12:00:00+00:00"
  }
}
```


Related Data Structures

Feature Group Version Response

Feature groups often output from pipelines:

```
{
  "success": true,
  "result": {
    "featureGroupVersion": "fg_version_123",
    "featureGroupId": "fg_456",
    "status": "COMPLETE",
    "dataMetrics": {
      "rowCount": 10000,
      "columnCount": 25
    },
    "schema": [
      {
        "name": "feature_1",
        "dataType": "float",
        "nullable": false
      }
    ]
  }
}
```

Batch Prediction Results

For pipelines generating predictions:

```
{
  "success": true,
  "result": {
    "batchPredictionVersion": "bp_version_123",
    "status": "COMPLETE",
    "downloadUrl": "https://...",
    "outputLocation": "s3://bucket/predictions.csv"
  }
}
```

Pipeline Object Classes

Based on the Python SDK documentation, key classes include:

Pipeline:
- `pipelineId`: Unique identifier
- `name`: Pipeline name
- `projectId`: Associated project
- `createdAt`: Creation timestamp

PipelineVersion:
- `pipelineVersionId`: Version identifier
- `pipelineId`: Parent pipeline ID
- `status`: Execution status (PENDING, RUNNING, COMPLETE, FAILED)
- `completedAt`: Completion timestamp

PipelineStepVersion:
- `stepId`: Step identifier
- `stepName`: Human-readable step name
- `status`: Step execution status
- `logs`: Execution logs
- `metrics`: Performance metrics

PipelineVersionLogs:
- `logs` : Complete log content
- `pipelineVersionId` : Associated version
- `timestamp` : Log timestamp

---

4. How to Query Pipeline Results by Pipeline ID

Step-by-Step Process

Step 1: Get Pipeline Information

Use the pipeline ID to retrieve basic pipeline information:

```
curl -H "apiKey: YOUR_API_KEY" \
     "https://api.abacus.ai/api/v0/describePipeline?pipelineId=PIPELINE_ID"
```

Response includes:
- Pipeline name and configuration
- Latest pipeline version ID
- Overall pipeline status

Step 2: Query Specific Pipeline Version

Use the pipeline version ID from Step 1 to get detailed execution results:

```
curl -H "apiKey: YOUR_API_KEY" \
     "https://api.abacus.ai/api/v0/describePipelineVersion?pipelineVer-
sionId=VERSION_ID"
```

Response includes:
- Execution status (PENDING, RUNNING, COMPLETE, FAILED)
- Start and completion timestamps
- Step-by-step results
- Metrics and outputs

Step 3: Retrieve Pipeline Execution Logs

Get detailed logs for troubleshooting or auditing:

```
curl -H "apiKey: YOUR_API_KEY" \
     "https://api.abacus.ai/api/v0/listPipelineVersionLogs?pipelineVer-
sionId=VERSION_ID"
```

Step 4: Access Step-Specific Results

For granular step-level information:

```
curl -H "apiKey: YOUR_API_KEY" \
     "https://api.abacus.ai/api/v0/getStepVersionLogs?stepVersionId=STEP_VERSION_ID"
```

Complete Python SDK Example

```python
from abacusai import ApiClient

# Initialize client with API key
client = ApiClient(api_key='YOUR_API_KEY')

# Step 1: Describe the pipeline
pipeline = client.describe_pipeline(pipeline_id='pipeline_123abc')
print(f"Pipeline Name: {pipeline.name}")
print(f"Latest Version: {pipeline.pipeline_version.pipeline_version_id}")

# Step 2: Get detailed version information
version = client.describe_pipeline_version(
    pipeline_version_id=pipeline.pipeline_version.pipeline_version_id
)
print(f"Status: {version.status}")
print(f"Completed At: {version.completed_at}")

# Step 3: Retrieve logs
logs = client.list_pipeline_version_logs(
    pipeline_version_id=version.pipeline_version_id
)
print(f"Logs: {logs.logs}")

# Step 4: Access step results
for step in version.steps:
    step_logs = client.get_step_version_logs(step_version_id=step.step_id)
    print(f"Step {step.step_name}: {step_logs}")
```

Querying Pipeline Outputs

If the pipeline generates feature groups or datasets:

```python
# Get feature group version from pipeline
feature_group = client.describe_feature_group(feature_group_id='fg_456')
latest_version = feature_group.latest_version

# Execute queries on the results
result = client.execute_feature_group_operation(
    feature_group_id='fg_456',
    sql="SELECT * FROM feature_group LIMIT 100"
)
```

Polling for Pipeline Completion

For asynchronous pipeline execution:

```python
import time

# Start pipeline execution
run_result = client.run_pipeline(pipeline_id='pipeline_123')
version_id = run_result.pipeline_version_id

# Poll until complete
while True:
    version = client.describe_pipeline_version(pipeline_version_id=version_id)

    if version.status in ['COMPLETE', 'FAILED']:
        print(f"Pipeline finished with status: {version.status}")
        break

    print(f"Current status: {version.status}")
    time.sleep(30)  # Wait 30 seconds before checking again

# Retrieve results after completion
if version.status == 'COMPLETE':
    logs = client.list_pipeline_version_logs(pipeline_version_id=version_id)
    print(f"Execution logs: {logs.logs}")
```

Batch Querying Multiple Pipeline Versions

```python
# List all versions of a pipeline
versions = client.list_pipeline_versions(pipeline_id='pipeline_123')

for version in versions:
    print(f"Version ID: {version.pipeline_version_id}")
    print(f"Status: {version.status}")
    print(f"Created: {version.created_at}")
    print(f"Completed: {version.completed_at}")
```

---

5. Rate Limits and Best Practices for API Integration

Rate Limits

ChatLLM API Limits

For ChatLLM services (which may be used in pipelines):
- **Generous base limits**: Thousands of messages without attachments before restrictions
- **Rate limiting triggers**: Large attachments automatically trigger rate limiting
- **Fallback mechanism**: System automatically switches to alternative LLMs to maintain service continuity
- **Token allocation**: Abacus.AI provides approximately **10x more tokens** than competing services

General API Limits

While specific rate limits for standard REST API endpoints are not explicitly documented, Abacus.AI implements:
- **Dynamic rate limiting** based on usage patterns and system capacity
- **Fair usage policies** to ensure resource availability for all users
- **No hard cap on users**, but billing tied to team size and usage

Best Practices for API Integration

1. Authentication & Security

DO:
- Store API keys in environment variables or secure vaults
- Use deployment tokens for deployment-specific operations to limit scope
- Rotate API keys periodically
- Delete compromised keys immediately

DON'T:
- Commit API keys to version control
- Share API keys across multiple applications
- Use API keys in client-side code

2. Error Handling & Retry Logic

Implement Exponential Backoff:

```python
import time
from random import uniform

def api_call_with_retry(func, max_retries=5):
    for attempt in range(max_retries):
        try:
            return func()
        except Exception as e:
            if attempt == max_retries - 1:
                raise

            # Exponential backoff with jitter
            wait_time = (2 ** attempt) + uniform(0, 1)
            print(f"Retry {attempt + 1} after {wait_time:.2f}s")
            time.sleep(wait_time)
```

Handle Common HTTP Error Codes:
- `400 (MissingParameterError)`: Check required parameters
- `403 (GenericPermissionDeniedError)`: Verify API key validity
- `404 (Generic404Error)`: Confirm resource IDs are correct
- `409 (ConflictError)`: Handle resource conflicts
- `424 (FailedDependencyError)`: Check dependent resources
- `429 (Rate Limit)`: Implement backoff and retry
- `5xx (NotReadyError)`: Wait and retry for server-side processing

3. Optimize Request Patterns

Batch Operations:

```python
# Instead of multiple individual calls
for item in items:
    client.predict(item)  # DON'T

# Use batch prediction endpoints
client.predict_multiple(items)  # DO
```

Reduce Attachment Sizes:
- Compress large files before upload

- Use streaming for large datasets
- Consider file connectors for very large data

Optimize Prompts (for LLM endpoints):
- Keep instructions concise
- Reduce `max_tokens` parameter to stay within limits
- Cache repeated requests when possible

4. Monitor and Log Usage

Track API Calls:

```
import logging
from datetime import datetime

def monitored_api_call(func, *args, **kwargs):
    start_time = datetime.now()

    try:
        result = func(*args, **kwargs)
        duration = (datetime.now() - start_time).total_seconds()

        logging.info(f"API call succeeded: {func.__name__}, duration: {duration}s")
        return result

    except Exception as e:
        duration = (datetime.now() - start_time).total_seconds()
        logging.error(f"API call failed: {func.__name__}, duration: {duration}s, er-
ror: {e}")
        raise
```

Monitor Billing and Usage:
- Check the billing dashboard regularly
- Set up alerts for unusual usage patterns
- Monitor team size and associated costs

5. Asynchronous Operations

Poll with Appropriate Intervals:

```
# For pipeline execution
POLL_INTERVAL = 30  # seconds
MAX_WAIT_TIME = 3600  # 1 hour

start_time = time.time()

while time.time() - start_time < MAX_WAIT_TIME:
    status = client.describe_pipeline_version(version_id)

    if status.status in ['COMPLETE', 'FAILED']:
        break

    time.sleep(POLL_INTERVAL)
```

Use Webhooks When Available:
- Set up webhooks for completion notifications instead of polling
- Reduces unnecessary API calls
- Faster response to status changes

6. Data Transfer Optimization

Upload Large Files Efficiently:

```
# Use multipart upload for large datasets
upload = client.create_dataset_version_from_upload('dataset_id')

with open('large_file.csv', 'rb') as f:
    upload.upload_file(f)

# Wait for processing to complete
dataset = client.describe_dataset(upload.dataset_id)
dataset.wait_for_inspection()
```

Download Results Efficiently:

```
# Download batch predictions
batch_prediction = client.describe_batch_prediction_version('bp_version_id')

with open('output.csv', 'wb') as output_file:
    batch_prediction.download_result_to_file(output_file)
```

7. Use SDK Over Direct REST Calls

Benefits:

- Built-in retry logic
- Automatic authentication handling
- Type checking and validation
- Better error messages
- Convenience methods (e.g., `wait_for_inspection()`)

Example:

```
# SDK approach (recommended)
from abacusai import ApiClient

client = ApiClient(api_key='YOUR_API_KEY')
project = client.describe_project(project_id='proj_123')

# Direct REST approach (more complex)
import requests

headers = {'apiKey': 'YOUR_API_KEY'}
response = requests.get(
    'https://api.abacus.ai/api/v0/describeProject',
    params={'projectId': 'proj_123'},
    headers=headers
)
```

8. Connection Management

Use Connection Pooling:

```python
import requests
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util.retry import Retry

session = requests.Session()

retry_strategy = Retry(
    total=3,
    backoff_factor=1,
    status_forcelist=[429, 500, 502, 503, 504]
)

adapter = HTTPAdapter(max_retries=retry_strategy)
session.mount("https://", adapter)

# Use session for all API calls
response = session.get(url, headers=headers)
```

9. Caching and Deduplication

Cache Static Responses:

```python
from functools import lru_cache

@lru_cache(maxsize=128)
def get_project_schema(project_id):
    return client.describe_project(project_id)

# Schema doesn't change frequently, so cache it
schema = get_project_schema('proj_123')
```

Deduplicate Requests:

- Track in-flight requests to avoid duplicate calls
- Use request IDs for idempotency when available

10. API Version Management

Stay Updated:

- Monitor API changelog for updates
- Test new endpoints in development before production
- Plan for deprecation notices
- Use versioned endpoints when available

Common Integration Patterns

Pattern 1: Batch Processing Pipeline

```
# 1. Upload data
upload = client.create_dataset_version_from_upload('dataset_id')
upload.upload_file(data_file)

# 2. Wait for processing
dataset = client.describe_dataset(upload.dataset_id)
dataset.wait_for_inspection()

# 3. Run pipeline
pipeline_run = client.run_pipeline(pipeline_id='pipeline_123')

# 4. Poll for completion
while True:
    version = client.describe_pipeline_version(pipeline_run.pipeline_version_id)
    if version.status == 'COMPLETE':
        break
    time.sleep(30)

# 5. Retrieve results
results = client.describe_feature_group_version(version.output_feature_group_id)
```

Pattern 2: Real-time Prediction Integration

```
# 1. Deploy model
deployment = client.create_deployment(
    model_id='model_123',
    name='Production Deployment'
)
client.start_deployment(deployment.deployment_id)

# 2. Make predictions
result = client.predict(
    deployment_id=deployment.deployment_id,
    deployment_token=deployment.deployment_token,
    query_data={'feature1': value1, 'feature2': value2}
)

# 3. Monitor performance
monitor = client.describe_realtime_monitor(deployment.deployment_id)
```

Pattern 3: Scheduled Pipeline Execution

```python
# 1. Create refresh schedule
schedule = client.create_refresh_policy(
    name='Daily Processing',
    cron='0 2 * * *'  # 2 AM daily
)

# 2. Attach to pipeline
client.set_refresh_schedule(
    pipeline_id='pipeline_123',
    refresh_policy_id=schedule.refresh_policy_id
)

# 3. Monitor executions
versions = client.list_pipeline_versions(pipeline_id='pipeline_123')
```

Summary

Key Takeaways

1. **Authentication**: Simple API key-based authentication using the `apiKey` header
2. **Pipeline Querying**: Use `describePipeline` and `describePipelineVersion` with pipeline IDs
3. **Response Format**: Consistent JSON structure with `success` and `result` fields
4. **Rate Limits**: Generous limits with automatic fallback mechanisms for LLM services
5. **Best Practices**: Use SDK, implement retry logic, optimize batch operations, monitor usage

Quick Reference

Base URL:

```
https://api.abacus.ai/api/v0/
```

Authentication Header:

```
apiKey: YOUR_API_KEY
```

Key Pipeline Endpoints:
- `POST /runPipeline` - Execute pipeline
- `GET /describePipelineVersion` - Get execution results
- `GET /listPipelineVersionLogs` - Retrieve logs

Python SDK Installation:

```
pip install abacusai
```

Basic SDK Usage:

```python
from abacusai import ApiClient

client = ApiClient(api_key='YOUR_API_KEY')
pipeline = client.describe_pipeline(pipeline_id='pipeline_id')
```

Additional Resources

- **Official API Reference**: https://abacus.ai/help/api/ref
- **API Documentation**: https://abacus.ai/help/api
- **Python SDK Classes**: https://abacus.ai/help/api/classes/
- **API Dashboard**: Generate and manage keys at https://abacus.ai/app/profile/apikey
- **Security Policy**: https://abacus.ai/security

Research compiled on: October 4, 2025