Table of Contents

Appendices

1. **Ardanis Cloud Support Services**

1.1 Service Overview

Ardanis provides cloud support services across public, private, and hybrid environments. Our services help organisations plan, migrate, secure, and operate cloud environments effectively. We support cloud adoption and ongoing operations through structured delivery, assurance, incident management, and operational support.

Services cover managed cloud operations, cloud migration, security services, financial management, quality assurance, testing, and optimisation, enabling customers to maintain efficient, secure, and optimised cloud environments.

1.2 Scope of Service: Our services include:

- **Managed Cloud Operations** – public, private, hybrid; IaaS, PaaS, SaaS; application management.
- **Cloud Migration Planning** – capability analysis, enterprise architecture, gap assessment, architecture options, and delivery road mapping.
- **Setup and Migration** – auditing, data/information structure design, engagement and communication planning, project management, mobilisation coordination, service optimisation.
- **Security Services** – strategy, risk management, design, incident management, audits, quality assurance/testing.
- **Quality Assurance & Testing** – performance, load, stress, scalability, capacity, operational readiness, and accessibility testing.
- **Cloud Financial Management** – FinOps and GreenOps services.
- **Ongoing Operational Support** – incident logging, reporting, proactive results analysis, operational assistance.

1.3 Service Features

- Cloud architecture design and technical assurance.
- Cloud migration planning and delivery.
- Managed public, private, and hybrid cloud operations.
- Cloud security strategy, design, and incident management.
- Financial management and sustainability (FinOps/GreenOps).
- Software development, design, delivery, and implementation.

- Test automation development and implementation.
- Performance, scalability, and capacity testing.
- Quality assurance and operational readiness testing.
- Incident logging, reporting, and ongoing operational support.

1.4 Service Benefits

- Accelerates cloud migration and reduces deployment times.
- Improves operational efficiency across cloud environments.
- Enhances security and reduces business risk.
- Optimises cloud costs and supports sustainable operations.
- Ensures high-quality software development and implementation.
- Minimises errors through test automation and QA processes.
- Improves system performance, scalability, and reliability.
- Supports ongoing operations with proactive incident management.
- Streamlines project delivery and governance.
- Reduces maintenance effort and improves service continuity.

1.5 Support & Service Management

- **Support Levels:** Level 1 (L1), Level 2 (L2), Level 3 (L3): adaptable to customer requirements.
- **Roles Provided:** Technical Account Manager and Cloud Support Engineers.
- **Email Support:** 99% of queries responded to within 30 minutes; weekend support by agreement.
- **Support Costs:** Depend on the deployed solution; typically, 10 – 15%.
- **Delivery Model:** Services are delivered on-site, remote, or hybrid.

1.6 Onboarding & Operational Readiness

Customers can engage Ardanis for:

- Initial migration planning and capability assessments.
- Mobilisation and governance coordination.
- Operational readiness and assurance testing.

1.7 Exit / Offboarding

Processes for service termination and data handling at the end of engagement will be agreed and documented with the customer as part of the onboarding process.

1.8 Security & Compliance

- Security strategy, risk management, and design.
- Security incident management, audits, and quality assurance/testing.
- Services delivered in alignment with customer cloud environments.

1.9 Constraints & Exclusions

- We do not resell third-party services.
- We do not provide end-user training.
- We do not provide penetration testing services.
- Phone or Web Chat Support is provided exceptionally by agreement.

1.10 Special Conditions / Optional Details

Optional details such as Business Continuity and Disaster Recovery, Specific Hosting Locations, Maintenance Windows, SLAs, KRAs, KPIs and any service credits are agreed with the customer on a case-by-case basis.

2. Software as a Service (SaaS)

2.1 Service Overview

Aileen is delivered as a Software as a Service (SaaS). It is an AI-powered orchestration solution that helps organisations automate, coordinate, and manage complex enterprise workflows. The service unifies systems, automates tasks, and provides actionable insights to improve decision-making, reduce errors, and support reliable business operations.

2.2 Service Features

- AI orchestration across people, systems, and enterprise data.
- Highly configurable workflows tailored to organisational processes.
- Intelligent automation of routine and complex workflows.
- LLM-powered capabilities using RAG, sentiment analysis, and translation.
- Integration with CRMs, ERPs, and third-party enterprise systems.
- Real-time workflow monitoring with operational transparency.
- Fully auditable system with comprehensive activity logging.
- Enterprise-grade security with GDPR-aligned compliance controls.
- Cloud-native solution supporting multiple environments.
- Scalable deployment from department-level to enterprise-wide use.

2.3 Service Benefits

- Streamlines processes and reduces manual effort.
- Improves decision-making speed and accuracy.
- Enhances operational efficiency across workflows.
- Extracts value from legacy or complex systems.
- Connects teams and data for transparent operations.
- Scales easily from small projects to enterprise-wide use.
- Supports measurable business value and ROI.
- Improves adaptability to changing business priorities.

- Enables faster, smarter, reliable enterprise operations.
- Reduces workflow bottlenecks and operational costs.

2.4 Onboarding & Offboarding

At the start of engagement, customers are provided guidance on accessing Aileen via the API. At contract end, customer-owned configurations can be extracted or deleted according to agreed offboarding processes.

Note: The Support & Service Management and Constraints & Exclusions described in the Cloud Support Services section also apply to Aileen/SaaS.

Appendix A: List of Cloud Support Services Offered:

Managed Cloud

Managed Private Cloud

Managed Private IaaS
Managed Private SaaS/PaaS
Application management

Managed Public Cloud

Managed Public IaaS
Managed Public SaaS/PaaS
Application management

Managed Hybrid Cloud

Managed Hybrid IaaS
Managed Hybrid SaaS/PaaS
Application management

Cloud Financial Management Services

FinOps Services
GreenOps Services

Cloud Migration Planning

Capability analysis
Enterprise architecture
Cloud gap assessment
Architecture options analysis
Delivery Road-mapping

Set Up and Migration

Data auditing and organising
Data/information structure design
Engagement and communication planning
Project management and governance
Service optimisation/right sizing
Mobilisation coordination

Security Services

Security strategy
Security risk management
Security design
Security incident management
Security audit services

Security quality assurance (QA) and testing

Security quality assurance (QA) and testing

Quality Assurance and Performance Testing

Test automation
Development
Implementation

Performance testing

Load Testing
Stress Testing
Volume Testing
Soak Testing
Scalability Testing
Capacity Planning

Assurance/testing design

Assurance/testing design

Infrastructure performance testing

Infrastructure performance testing

Operational readiness testing

Accessibility testing

Ongoing Support

Product support capabilities
Logging and management of incidents
Reporting and proactive results analysis
Dispatch of service technicians and/or parts

End user training coordination

Appendix B: List of SaaS, Services Offered

Managed Private Cloud
FinOps Services
Capability analysis
Enterprise architecture
Cloud gap assessment
Architecture options analysis
Delivery Road-mapping
Data auditing and organising
Data/information structure design
Engagement and communication planning
Project management and governance
Service optimisation/right sizing
Mobilisation coordination
Security design
Test automation
Operational readiness testing
Basic troubleshooting skills
Advanced troubleshooting skills
Logging and management of incidents
Reporting and proactive results analysis
AI software services
Data integration and intelligence
Development languages, environments and tools
Software change, configuration and process management
Email
Team collaboration
Financial
Customer service
Contact centre

Appendix C: Webhook Flow Execution Guide

Overview

This guide describes how external systems can execute Aileen flows via webhook endpoints, including real-time streaming capabilities. The webhook system provides a secure, scalable way for external applications to trigger flow executions and receive streaming data in real-time.

Architecture

The webhook flow execution system consists of two main components:

1. **Webhook Endpoint** (POST /flows/webhook) - Initiates flow execution
2. **Streaming Endpoint** (GET /flows/flow-streaming/:flowRunIdentifier) - Provides real-time data streaming

Authentication

Webhook Auth Token

External systems must authenticate using a **Webhook Auth Token** that contains:

- **Flow ID**: The specific flow to be executed
- **Company ID**: The company context for the flow
- **HTTP Verb**: Allowed HTTP method (typically POST)
- **Callback URL**: Optional callback URL for notifications
- **Headers**: Optional custom headers for callbacks
- **Connection Credentials**: Optional webhook connection credentials

Token Structure

```
type WebhookAuthToken = {

 id: string;

 companyId: string;

 flowId: string;

 httpVerb: string;



 callbackUrl?: string;

 headers?: string;

 webhookConnectionCredsId?: string;
```

token: string; // The actual bearer token

valid: boolean;

title: string;

description?: string;

webhookConnectionCreds?: WebhookConnectionCreds;

};

Token Creation

Webhook tokens are created on the Aileen side through the admin interface or API:

- **Endpoint**: POST /webhook-auth-tokens
- **Required Permission**: Webhook permission
- **Token Generation**: Tokens are automatically generated and encrypted

Authentication Header

Include the token in the Authorization header:

Authorization: Bearer <webhook_token>

Flow Execution

Webhook Endpoint

Endpoint: POST /flows/webhook

Headers:

Authorization: Bearer <webhook_token>

Content-Type: application/json

Request Payload

The request body must conform to the WebhookRequestPayloadDto structure:

type WebhookRequestPayloadDto = {

interactionSourceId: string | number; // Unique identifier for this interaction

interactionSource: string; // Source system identifier

interactionType: string; // Type of interaction

payload: Record<string, string | number | boolean | object>; // Flow input data

};

Flow Input Values

The payload property contains the actual data to be passed to the flow. Values should be provided directly as their primitive types or objects:

- **Strings**: Pass as string literals
- **Numbers**: Pass as numeric values
- **Booleans**: Pass as boolean values (true/false)
- **Objects**: Pass as JSON objects (will be automatically stringified)

The system will automatically detect the type and convert values to the internal FlowManagerInputValuesDto format.

Example Request

```
{
 "interactionSourceId": "user-12345",
 "interactionSource": "external-system",
 "interactionType": "document-analysis",
 "payload": {
  "document": "This is the document content to analyze",
  "analysisType": "sentiment",
  "priority": 1,
  "metadata": {
   "source": "api",
   "timestamp": "2024-01-15T10:30:00Z"
  }
 }
}
```

Response

The webhook endpoint returns a **flow run identifier**:

"550e8400-e29b-41d4-a716-446655440000"

This identifier is used to subscribe to streaming data for this specific flow execution.

Real-Time Streaming

Streaming Endpoint

Endpoint: GET /flows/flow-streaming/:flowRunIdentifier

Headers:

Authorization: Bearer <webhook_token>

Streaming Protocol

The streaming endpoint uses **Server-Sent Events (SSE)** with the following characteristics:

- **Content-Type**: text/plain
- **Transfer-Encoding**: chunked
- **Connection**: keep-alive
- **Cache-Control**: no-cache

Streaming Data Format

The stream sends data in the following format:

data: <streamed_content>

Where <streamed_content> is the actual tokenized data from the flow execution.

Streaming Events

The streaming system supports two types of events:

1. **Streaming Event**: Contains actual data tokens

typescript { type: 'streaming', data: { nodeId: string; streamedData: string; } }

1. **Completion Event**: Indicates the flow has finished typescript { type: 'completed'; }

Example Streaming Session

```
const eventSource = new EventSource('/flows/flow-streaming/550e8400-e29b-41d4-a716-446655440000', {
 headers: {
  'Authorization': 'Bearer <webhook_token>',
 },
```

```
});

eventSource.onmessage = function (event) {

 console.log('Received data:', event.data);

};

eventSource.onerror = function (event) {

 console.error('Streaming error:', event);

};
```

Workflow Patterns

Chat-Style Interaction

External systems can build chat-like workflows by:

1. **Sending Multiple Requests**: Use the same interactionSourceId across multiple webhook calls
2. **Subscribing to Callbacks**: Each request returns a unique flowRunIdentifier
3. **Associating Callbacks**: Link callbacks to interactions using the interactionSourceId

Example Chat Workflow

```
class ChatWorkflow {

 constructor(interactionId, webhookToken) {

  this.interactionId = interactionId;

  this.webhookToken = webhookToken;

  this.activeStreams = new Map();

 }

 async sendMessage(message) {

  // Send webhook request

  const response = await fetch('/flows/webhook', {

   method: 'POST',

   headers: {

    'Authorization': `Bearer ${this.webhookToken}`,
```

```
    'Content-Type': 'application/json',
   },
   body: JSON.stringify({
    interactionSourceId: this.interactionId,
    interactionSource: 'chat-system',
    interactionType: 'user-message',
    payload: {
     message: message,
    },
   }),
  });
  const flowRunIdentifier = await response.text();
  // Subscribe to streaming
  this.subscribeToStream(flowRunIdentifier);
  return flowRunIdentifier;
 }
 subscribeToStream(flowRunIdentifier) {
  const eventSource = new EventSource(`/flows/flow-streaming/${flowRunIdentifier}`, {
   headers: {
    'Authorization': `Bearer ${this.webhookToken}`,
   },
  });
  this.activeStreams.set(flowRunIdentifier, eventSource);
  eventSource.onmessage = (event) => {
   this.handleStreamingData(flowRunIdentifier, event.data);
  };
```

```
  eventSource.onerror = (event) => {
    this.handleStreamingError(flowRunIdentifier, event);
    this.activeStreams.delete(flowRunIdentifier);
  };
 }
 handleStreamingData(flowRunIdentifier, data) {
  // Process streaming data
  console.log(`Stream ${flowRunIdentifier}: ${data}`);
 }

 handleStreamingError(flowRunIdentifier, error) {
  console.error(`Stream ${flowRunIdentifier} error:`, error);
 }
 cleanup() {
  this.activeStreams.forEach((stream) => {
    stream.close();
  });
  this.activeStreams.clear();
 }
}
```

Error Handling

Webhook Errors

- **401 Unauthorized**: Invalid or missing webhook token
- **400 Bad Request**: Invalid payload structure
- **404 Not Found**: Flow not found or not accessible
- **429 Too Many Requests**: Rate limiting applied

Streaming Errors

- **404 Not Found**: Invalid flow run identifier
- **Connection Drops**: Automatic reconnection not supported; client must handle reconnection

Best Practices

1. **Token Management**: Store webhook tokens securely
2. **Error Handling**: Implement proper error handling for both webhook and streaming endpoints
3. **Connection Management**: Properly close streaming connections to prevent resource leaks
4. **Rate Limiting**: Respect rate limits and implement exponential backoff
5. **Payload Validation**: Validate payload structure before sending requests

Security Considerations

1. **Token Security**: Webhook tokens are encrypted and should be treated as sensitive credentials
2. **HTTPS Only**: Always use HTTPS for webhook communications
3. **Access Control**: Webhook tokens are scoped to specific flows and companies
4. **Audit Logging**: All webhook executions are logged for audit purposes

Appendix D: Flow Streaming API Guide

This guide describes how to execute flows and consume streaming data using the Flow Streaming API.

Overview

The Flow Streaming API allows you to:

1. Execute a flow via webhook endpoint
2. Consume real-time streaming data from the flow execution
3. Choose between two streaming formats: V1 (plain text) and V2 (SSE JSON)

Endpoints

1. Execute Flow (Webhook)

Endpoint: POST /api/v1/flows/webhook

Authentication: Bearer token required

Description: Initiates a flow execution and returns a flowRunIdentifier that can be used to consume the streaming data.

Request Headers:

Authorization: Bearer <token>

Content-Type: application/json

Request Body:

```
{
 "interactionSourceId": "30",
 "interactionSource": "freshdesk",
 "interactionType": "ticket",
 "payload": {
  "topic": "blue sky"
 }
}
```

Response:

- **Status:** 200 OK
- **Content-Type:** text/plain
- **Body:** flowRunIdentifier (UUID string)

The flowRunIdentifier is returned as plain text and must be used to consume the stream.

Example:

```
const response = await fetch('http://localhost:8080/api/v1/flows/webhook', {
 method: 'POST',
 headers: {
  'Content-Type': 'application/json',
  'Authorization': 'Bearer <token>',
 },
 body: JSON.stringify({
  interactionSourceId: '30',
  interactionSource: 'freshdesk',
  interactionType: 'ticket',
  payload: {
   topic: 'blue sky',
  },
 }),
});


const flowRunIdentifier = await response.text();
console.log('Flow Run Identifier:', flowRunIdentifier);
```

2. Consume Flow Stream

Endpoint: GET /api/v1/flows/flow-streaming/:flowRunIdentifier

Authentication: Bearer token required

Query Parameters:

- version (optional): 'v1' or 'v2' (default: 'v1')

Description: Streams real-time data from the flow execution. Supports two formats:

- **V1**: Plain text chunks for streaming node content
- **V2**: Server-Sent Events (SSE) with structured JSON messages

Request Headers:

Authorization: Bearer <token>

Streaming Formats

V1 Format (Plain Text)

Content-Type: text/plain **Transfer-Encoding:** chunked

Description: V1 streams only the content from nodes with nodeMessageCategory: 'streaming'. It sends plain text chunks directly without any JSON structure.

What is streamed:

- Only content field from NodeStreamingMessage where nodeMessageCategory === 'streaming'
- Stream ends when flow completes or errors

Example:

```
async function consumeV1Stream(flowRunIdentifier, bearerToken) {

 const response = await fetch(


` http://localhost:8080/api/v1/flows/flow-streaming/${flowRunIdentifier}?version=v1` ,

  {

    method: 'GET',

    headers: {

     'Authorization': bearerToken,

    },

  },
```

```
);

 const reader = response.body.getReader();
 const decoder = new TextDecoder();
  while (true) {
   const { value, done } = await reader.read();
   if (done) break;

   const chunk = decoder.decode(value, { stream: true });
   if (chunk) {
    console.log('Received chunk:', chunk);
    // Process plain text chunk
   }
  }
}
```

Response Handling:

- Read chunks as they arrive
- Each chunk is plain text content
- No JSON parsing needed
- Stream ends when connection closes

V2 Format (SSE JSON)

Content-Type: text/event-stream; charset=utf-8

Description: V2 streams structured JSON messages using Server-Sent Events (SSE) format. It includes all message types: flow messages, node actions, and node streaming messages.

Format: Each event follows the SSE specification:

data: {json_object}\n\n

Initial Message: The first message sent is a status indicator:

data: {"status": "start", "version": "v2"}\n\n

Completion Marker: When the stream completes, a special marker is sent:

data: [DONE]\n\n

Example:

```
async function consumeV2Stream(flowRunIdentifier, bearerToken) {
 const response = await fetch(
  `http://localhost:8080/api/v1/flows/flow-streaming/${flowRunIdentifier}?version=v2`,
  {
   method: 'GET',
   headers: {
    'Authorization': bearerToken,
   },
  },
 );
 const reader = response.body.getReader();
 const decoder = new TextDecoder();
 let buffer = '';
 while (true) {
  const { value, done } = await reader.read();
  if (done) break;
  buffer += decoder.decode(value, { stream: true });
  // Process complete SSE events (separated by \n\n)
  let separatorIndex;
  while ((separatorIndex = buffer.indexOf('\n\n')) !== -1) {
   const event = buffer.slice(0, separatorIndex);
```

```
   buffer = buffer.slice(separatorIndex + 2);
   processSSEEvent(event);
  }
}
// Process any remaining buffer
if (buffer.trim()) {
  processSSEEvent(buffer);
}
function processSSEEvent(rawEvent) {
  const lines = rawEvent.split(/\r?\n/);
  const dataLines = [];
  for (const line of lines) {
   if (line.startsWith('data:')) {
     const data = line.slice(5).trim();
     // Check for completion marker
     if (data === '[DONE]') {
       console.log('Stream completed');
       return;
     }
     dataLines.push(data);
   }
  }
  if (dataLines.length === 0) return;

  const jsonText = dataLines.join('\n').trim();
  if (!jsonText) return;
```

```
  try {
    const message = JSON.parse(jsonText);
    console.log('Received message:', message);
    // Process message based on its category and type
  } catch (error) {
    console.error('Failed to parse JSON:', error, jsonText);
  }
 }
}
```

Response Handling:

1. Read chunks incrementally
2. Buffer until you find \n\n separator
3. Extract data: lines from each event
4. Parse JSON from the data line
5. Handle [DONE] marker to detect completion

Data Structures

V2 Message Types

All V2 messages extend a base structure and can be one of three types:

Type Definitions (TypeScript)

```
// Base types
type MessageCategory = 'flow' | 'node';
type NodeMessageCategory = 'action' | 'streaming';
type MessageStage = 'progress' | 'complete' | 'error';
type NodePhase = 'thinking' | 'streaming_output' | 'done';


// Base message interface
interface BaseMessage {
  category: MessageCategory;
```

```
  flowId: string; // logical flow id
  seq?: number; // monotonic sequence number across the stream
  timestamp: string; // ISO-8601 timestamp

  metadata?: Record<string, unknown>;
  producer?: string; // service/node that emitted the event
  flowRunIdentifier: string; // unique identifier for this flow run
  error?: { code?: string; message: string; details?: unknown } | string;
}
// Flow-level messages
interface FlowMessage extends BaseMessage {
  category: 'flow';
  stage: MessageStage; // 'progress' | 'complete' | 'error'
}
// Node action messages (single non-streaming events)
interface NodeActionMessage extends BaseMessage {
  category: 'node';
  nodeMessageCategory: 'action';
  nodeId: string;
  nodeLabel?: string;
  nodePhase?: NodePhase;
  stage: MessageStage;
  content?: string | Record<string, unknown>;
  result?: Record<string, unknown>;
  error?: { code?: string; message: string; details?: unknown } | string;
}
```

```
// Node streaming messages (incremental content)
interface NodeStreamingMessage extends BaseMessage {
  category: 'node';
  nodeMessageCategory: 'streaming';
  nodeId: string;
  nodeLabel?: string;
  nodePhase?: NodePhase;
  stage: MessageStage;
  seq: number; // required for streaming chunks ordering
  contentType: 'text' | 'json' | 'binary' | string;
  content?: string | Record<string, unknown>;
  partIndex?: number; // identifies a logical part to patch
  op?: 'append' | 'replace' | 'set'; // operation type
  final?: boolean; // optional final flag for streaming phases
}
// Union type for all streamed messages
type StreamedData = FlowMessage | NodeStreamingMessage | NodeActionMessage;
```

Message Examples

Flow Message (Progress)

```
{
  "category": "flow",
  "flowId": "flow-123",
  "flowRunIdentifier": "550e8400-e29b-41d4-a716-446655440000",
 "stage": "progress",
  "timestamp": "2024-01-15T10:30:00.000Z",
```


```
 "seq": 1
}
```


Flow Message (Complete)


```
{
 "category": "flow",
 "flowId": "flow-123",
 "flowRunIdentifier": "550e8400-e29b-41d4-a716-446655440000",
 "stage": "complete",
 "timestamp": "2024-01-15T10:35:00.000Z",
 "seq": 100
}
```


Flow Message (Error)


```
{
 "category": "flow",
 "flowId": "flow-123",
 "flowRunIdentifier": "550e8400-e29b-41d4-a716-446655440000",
 "stage": "error",
 "timestamp": "2024-01-15T10:32:00.000Z",
 "error": {
  "code": "EXECUTION_ERROR",
  "message": "Node execution failed",
  "details": {}
 }
}
```


Node Action Message


```
{
```


```
 "category": "node",
 "nodeMessageCategory": "action",
 "nodeId": "node-456",
 "nodeLabel": "Analysis Node",
 "nodePhase": "done",
 "stage": "complete",
 "flowId": "flow-123",
 "flowRunIdentifier": "550e8400-e29b-41d4-a716-446655440000",
 "timestamp": "2024-01-15T10:31:00.000Z",
 "content": {
  "action": "analysis_complete",
  "summary": "Analysis completed successfully"
 },
 "result": {
  "output": "Analysis result data"
 },
 "seq": 50
}
```


Node Streaming Message


```
{
 "category": "node",
"nodeMessageCategory": "streaming",
 "nodeId": "node-789",
 "nodeLabel": "LLM Node",
 "nodePhase": "streaming_output",
 "stage": "progress",
```


```
 "flowId": "flow-123",
 "flowRunIdentifier": "550e8400-e29b-41d4-a716-446655440000",
 "timestamp": "2024-01-15T10:30:30.000Z",
 "seq": 25,
 "contentType": "text",
 "content": "This is a chunk of streaming content...",
 "op": "append"
}
```


V1 Data Structure

V1 does not send structured messages. It only streams the content field from NodeStreamingMessage objects as plain text chunks. There is no JSON structure or metadata in V1 format.

Example V1 Chunk:

This is a chunk of streaming content...

Complete Example

Here's a complete example that demonstrates the full flow:


```
const BEARER_TOKEN = 'Bearer <your-token>';
const BASE_URL = 'http://localhost:8080/api/v1';
// Step 1: Execute flow via webhook
async function executeFlow() {
 const webhookPayload = {
  interactionSourceId: '30',
  interactionSource: 'freshdesk',
  interactionType: 'ticket',
  payload: {
   topic: 'blue sky',
  },
```

```
};
const webhookResponse = await fetch(`${BASE_URL}/flows/webhook`, {
 method: 'POST',
 headers: {
  'Content-Type': 'application/json',
  'Authorization': BEARER_TOKEN,
 },
 body: JSON.stringify(webhookPayload),
});
if (!webhookResponse.ok) {
 throw new Error(`Webhook failed: ${webhookResponse.status}`);
}
const flowRunIdentifier = await webhookResponse.text();
console.log('Flow started with identifier:', flowRunIdentifier);
return flowRunIdentifier;
}
// Step 2: Consume stream (V2 example)
async function consumeStreamV2(flowRunIdentifier) {
 const streamUrl = `${BASE_URL}/flows/flow-streaming/${flowRunIdentifier}?version=v2`;
 const response = await fetch(streamUrl, {
  method: 'GET',
  headers: {
   'Authorization': BEARER_TOKEN,
  },
 });
 if (!response.ok) {
```

```
  throw new Error(` Stream failed: ${response.status}` );
 }
 const reader = response.body.getReader();
 const decoder = new TextDecoder();
 let buffer = '';
while (true) {
  const { value, done } = await reader.read();
  if (done) break;
  buffer += decoder.decode(value, { stream: true });
  let separatorIndex;
  while ((separatorIndex = buffer.indexOf('\n\n')) !== -1) {
   const event = buffer.slice(0, separatorIndex);
   buffer = buffer.slice(separatorIndex + 2);
   await processSSEEvent(event);
  }
 }
 if (buffer.trim()) {
  await processSSEEvent(buffer);
 }
 async function processSSEEvent(rawEvent) {
  const lines = rawEvent.split(/\r?\n/);
  const dataLines = [];
  for (const line of lines) {
   if (line.startsWith('data:')) {
 const data = line.slice(5).trim();
    if (data === '[DONE]') {
```

```
    console.log('Stream completed');
    return;
   }
   dataLines.push(data);
 }
}

if (dataLines.length === 0) return;
 const jsonText = dataLines.join('\n').trim();
if (!jsonText) return;
try {
 const message = JSON.parse(jsonText);
 // Handle different message types
 switch (message.category) {
  case 'flow':
   if (message.stage === 'complete') {
    console.log('Flow completed');
   } else if (message.stage === 'error') {
    console.error('Flow error:', message.error);
   }
   break;
  case 'node':
   if (message.nodeMessageCategory === 'streaming') {
    // Handle streaming content
    if (typeof message.content === 'string') {
     process.stdout.write(message.content);
```

```
        }
       } else if (message.nodeMessageCategory === 'action') {
        // Handle node action
        console.log('Node action:', message.nodeLabel, message.result);
       }
       break;
     }
   } catch (error) {
     console.error('Failed to parse JSON:', error, jsonText);
   }
 }
}
// Main execution
async function main() {
 try {
   const flowRunIdentifier = await executeFlow();
   await consumeStreamV2(flowRunIdentifier);
} catch (error) {
   console.error('Error:', error);
 }
}
 main();
```

Error Handling

Webhook Errors

- **400 Bad Request**: Invalid payload
- **401 Unauthorized**: Missing or invalid Bearer token
- **500 Internal Server Error**: Server error during flow initialization

Streaming Errors

- **404 Not Found**: flowRunIdentifier not found or flow hasn't started yet
- **401 Unauthorized**: Missing or invalid Bearer token
- **400 Bad Request**: Invalid version parameter

Note: If the flow has already completed before you connect to the stream, you may receive a 404 or the stream may immediately return all buffered messages and close.

Best Practices

1. **Start streaming immediately after webhook call**: The flow execution starts asynchronously, so connect to the stream as soon as possible to capture all messages.
2. **Handle connection errors**: Implement retry logic for network errors, but be aware that the stream may have progressed while you were disconnected.
3. **Use V2 for full context**: V2 provides complete message metadata and is recommended for most use cases.
4. **Use V1 for simple text streaming**: V1 is useful when you only need the streaming text content without metadata.
5. **Clean up connections**: Always close the stream connection when done or when errors occur to free server resources.
6. **Handle [DONE] marker**: In V2, always check for the [DONE] marker to know when the stream has completed.
7. **Monitor flow stages**: Check message.stage to track flow progress (progress, complete, error).

Notes

- The flowRunIdentifier is valid for 5 minutes after the flow completes (TTL cleanup)
- Multiple clients can connect to the same flowRunIdentifier to receive the same stream
- If you connect after the flow has started, you'll receive all previously buffered messages first
- The stream automatically closes when the flow completes or errors