Tutorial 19: Artifacts and Files - Content Generation and Management

Difficulty: advanced
Reading Time: 1.5 hours
Tags: advanced, artifacts, files, content-generation, file-management
Description: Generate and manage files, documents, and artifacts using agents for content creation, code generation, and file processing workflows.

:::info Verified Against Official Sources

This tutorial has been verified against the official ADK Python source code and documentation. All API calls, version numbering, and examples are accurate as of October 2025.

Verification Date: October 10, 2025
ADK Version: 1.16.0+

Implementation Note: The reference implementation uses async tools with `ToolContext` and the correct `artifact=` parameter (not `part=`). All artifacts are saved and retrieved successfully. See the "Troubleshooting" section for important notes about the Artifacts tab UI display.

:::

:::warning Important: Artifacts Tab UI Limitation

When using `InMemoryArtifactService` for local development, **the Artifacts tab in the web UI will appear empty**. This is expected behavior and does NOT mean your artifacts aren't working.

Your artifacts ARE being saved correctly! Access them via:
- ✅ Blue artifact buttons in chat (primary method)
- ✅ Ask agent "Show me all saved artifacts"
- ✅ Check server logs for HTTP 200 responses

See the [Troubleshooting section](#9-troubleshooting) for complete details.

:::

Tutorial 19: Artifacts & File Management

[View Implementation](./../../tutorial_implementation/tutorial19)

Goal: Master artifact storage, versioning, and retrieval to enable agents to create, manage, and track files across sessions, providing persistent state and audit trails.

Prerequisites:

- Tutorial 01 (Hello World Agent)
- Tutorial 08 (State & Memory)
- Tutorial 09 (Callbacks & Guardrails)
- Understanding of file I/O operations

What You'll Learn:

- Using `save_artifact()` to store files with versioning
- Retrieving artifacts with `load_artifact()`
- Listing all artifacts with `list_artifacts()`
- Managing credentials with `save_credential()` and `load_credential()`
- Building document processors with artifact tracking
- Implementing file provenance and audit trails
- Best practices for production artifact management

Time to Complete: 45-60 minutes

Why Artifacts Matter

Problem: Agents need to create and persist files (reports, data, images) across sessions, with version control and audit trails.

Solution: **Artifacts** provide structured file storage with automatic versioning and metadata tracking.

Benefits:

- 💾 **Persistence**: Files survive across agent sessions
- 📝 **Versioning**: Automatic version tracking for every save
- 🔍 **Discoverability**: List and search all artifacts
- 📊 **Audit Trail**: Track who created what and when
- 🔐 **Credentials**: Secure storage for API keys and tokens
- 🎯 **Context**: Agents can reference previously created files

Use Cases:

- Report generation and archival
- Data processing pipelines
- Document creation workflows
- File transformation chains
- Audit and compliance logging

1. Artifact Basics

What is an Artifact?

An **artifact** is a versioned file stored by the agent system. Each save creates a new version, and all versions are retained.

Source: `google/adk/agents/callback_context.py`, `google/adk/tools/tool_context.py`

Artifact Properties:

- **Filename**: Unique identifier
- **Version**: Auto-incrementing integer starting at 0 (0, 1, 2, ...)
- **Content**: The actual file data (as `types.Part`)
- **Metadata**: Author, timestamp, context

```
Artifact Structure:

+--------------------------------------------------------------------+
|                          ARTIFACT SYSTEM                           |
+--------------------------------------------------------------------+
|                                                                    |
|  +---------------------+     +---------------------+               |
|  |   Filename          |     |   Version History   |               |
|  |  "report.txt"       |---->|   v0, v1, v2, ...   |               |
|  +---------------------+     +---------------------+               |
|            |                            |                          |
|            v                            v                          |
|  +---------------------+     +---------------------+               |
|  |   Content           |     |   Metadata          |               |
|  |  types.Part         |     |  Author, Timestamp  |               |
|  |  (text/binary)      |     |  Context Info       |               |
|  +---------------------+     +---------------------+               |
|                                                                    |
+--------------------------------------------------------------------+
```

:::info Version Numbering
Artifact versions are **0-indexed**. The first save returns version 0, the second returns version 1, and so on.
:::

Implementation Note: Async Tools with ToolContext

All artifact operations are asynchronous. When building tools that use artifacts, they must be async functions that accept `ToolContext`:

```python
from google.adk.tools.tool_context import ToolContext
from google.genai import types

async def my_tool(param: str, tool_context: ToolContext) -> dict:
    """Tool that saves artifacts."""

    # Create artifact content
    content = f"Processed: {param}"
    artifact_part = types.Part.from_text(text=content)

    # Save artifact (note: 'artifact' parameter, not 'part')
    version = await tool_context.save_artifact(
        filename='output.txt',
        artifact=artifact_part  # Correct parameter name
    )

    return {
        'status': 'success',
        'report': f'Saved as version {version}',
        'data': {'version': version, 'filename': 'output.txt'}
    }
```

Key points:

- ✓ Use `async def` for tool functions
- ✓ Accept `tool_context: ToolContext` parameter
- ✓ Use `await` with `save_artifact()`, `load_artifact()`, `list_artifacts()`
- ✓ Use `artifact=` parameter (not `part=`) in ADK 1.16.0+
- ✓ Return structured dict with `status`, `report`, and `data` fields

Where Artifacts are Available

Artifacts can be accessed in:

```
Artifact Access Points:

+----------------------------------------------------------------------+
|                       ARTIFACT API ACCESS                            |
+----------------------------------------------------------------------+
|                                                                      |
|  +------------------------------+  +------------------------------+  |
|  |     CallbackContext          |  |       ToolContext            |  |
|  |    (Agent Callbacks)         |  |    (Function Tools)          |  |
|  +------------------------------+  +------------------------------+  |
|  | - save_artifact()            |  | - save_artifact()            |  |
|  | - load_artifact()            |  | - load_artifact()            |  |
|  | - list_artifacts()           |  | - list_artifacts()           |  |
|  +------------------------------+  +------------------------------+  |
|               |                              |                       |
|               +------------------------------+                       |
|                              |                                       |
|                              v                                       |
|                   +-----------------------+                          |
|                   |  Artifact Service     |                          |
|                   |  (Storage Backend)    |                          |
|                   +-----------------------+                          |
|                                                                      |
+----------------------------------------------------------------------+
```

```python
# 1. Callback context
from google.adk.agents import CallbackContext

async def my_callback(context: CallbackContext):
    # save, load, list artifacts
    version = await context.save_artifact('file.txt', part)
    artifact = await context.load_artifact('file.txt')
    files = await context.list_artifacts()

# 2. Tool context
from google.adk.tools.tool_context import ToolContext

async def my_tool(query: str, tool_context: ToolContext):
    # save, load, list artifacts
    version = await tool_context.save_artifact('data.json', part)
    artifact = await tool_context.load_artifact('data.json')
    files = await tool_context.list_artifacts()
```

Configuring Artifact Storage

Before using artifacts, configure an artifact service in your Runner:

```
Storage Configuration Architecture:

+-----------------------------------------------------------------------+
|                            RUNNER                                     |
+-----------------------------------------------------------------------+
|                                                                       |
|  +------------------------+                                           |
|  |       Agent            |                                           |
|  |  (uses artifacts)      |                                           |
|  +------------------------+                                           |
|              |                                                        |
|              v                                                        |
|  +------------------------+        +---------------------------+      |
|  |  Session Service       |        |  Artifact Service         |      |
|  |  (state management)    |        |  (file storage)           |      |
|  +------------------------+        +---------------------------+      |
|                                               |                       |
|              +--------------------------------+                       |
|              |                                |                       |
|              v                                v                       |
|   +----------------------+        +-------------------------+         |
|   | InMemoryArtifact     |        | GcsArtifactService      |         |
|   | Service              |        | (Google Cloud Storage)  |         |
|   | (dev/testing)        |        | (production)            |         |
|   +----------------------+        +-------------------------+         |
|                                                                       |
+-----------------------------------------------------------------------+
```

```python
from google.adk.runners import Runner
from google.adk.artifacts import InMemoryArtifactService, GcsArtifactService
from google.adk.sessions import InMemorySessionService
from google.adk.agents import Agent

# Option 1: In-Memory Storage (development/testing)
artifact_service = InMemoryArtifactService()

# Option 2: Google Cloud Storage (production)
# artifact_service = GcsArtifactService(bucket_name='your-gcs-bucket')

# Create agent
agent = Agent(
    name='my_agent',
    model='gemini-2.0-flash',
    # ... other config
)

# Configure runner with artifact service
runner = Runner(
    agent=agent,
    app_name='my_app',
    session_service=InMemorySessionService(),
    artifact_service=artifact_service  # Enable artifact storage
)
```

:::warning Required Configuration
If `artifact_service` is not configured, calling artifact methods will raise a `ValueError`. Always configure the artifact service before using artifacts.
:::

2. Saving Artifacts

Basic Save

```python
from google.adk.agents import CallbackContext
from google.genai import types

async def create_report(context: CallbackContext):
    """Create and save a report artifact."""

    # Create report content
    report_text = """
# Sales Report Q3 2025

Total Revenue: $1,245,000
Growth: +15% YoY

Top Products:
1. Product A: $450,000
2. Product B: $320,000
3. Product C: $275,000
    """.strip()

    # Create Part from text
    report_part = types.Part.from_text(report_text)

    # Save as artifact
    version = await context.save_artifact(
        filename='sales_report_q3_2025.md',
        part=report_part
    )

    print(f"Report saved as version {version}")
    return version
```

Save with Binary Data

```python
async def save_image(context: CallbackContext, image_bytes: bytes):
    """Save image artifact."""

    # Create Part from bytes
    image_part = types.Part(
        inline_data=types.Blob(
            data=image_bytes,
            mime_type='image/png'
        )
    )

    # Save image
    version = await context.save_artifact(
        filename='chart.png',
        part=image_part
    )

    return version
```

Versioning Behavior

```
Artifact Versioning Timeline:

Time: t0                  t1                  t2                  t3
      |                   |                   |                   |
      v                   v                   v                   v
+-------------+     +-------------+     +-------------+     +-------------+
|   Save 1    |     |   Save 2    |     |   Save 3    |     |   Save 4    |
| Version 0   |---->| Version 1   |---->| Version 2   |---->| Version 3   |
+-------------+     +-------------+     +-------------+     +-------------+
| "Draft"     |     | "Revised"   |     | "Final"     |     | "Updated"   |
+-------------+     +-------------+     +-------------+     +-------------+
      |                   |                   |                   |
      +-------------------+-------------------+-------------------+
                              |
                              v
                  +-------------------------+
                  | All Versions Retained   |
                  | Can Load Any Version    |
                  | (0, 1, 2, 3, ...)       |
                  +-------------------------+
```

```python
# First save - creates version 0
v1 = await context.save_artifact('report.txt', part1)
print(v1)  # Output: 0

# Second save - creates version 1
v2 = await context.save_artifact('report.txt', part2)
print(v2)  # Output: 1

# Third save - creates version 2
v3 = await context.save_artifact('report.txt', part3)
print(v3)  # Output: 2

# All versions retained and accessible (0, 1, 2, ...)
```

3. Loading Artifacts

```
Artifact Lifecycle Operations:

+---------------------------------------------------------------------+
|                    ARTIFACT OPERATIONS                              |
+---------------------------------------------------------------------+
|                                                                     |
|  +-----------------+     +------------------+     +---------------+ |
|  |  save_artifact  |---->| Artifact Storage |---->| Returns       | |
|  |  (filename,     |     | (all versions)   |     | Version       | |
|  |   content)      |     |                  |     | Number        | |
|  +-----------------+     +------------------+     +---------------+ |
|                                  |                                  |
|                                  |                                  |
|  +-----------------+             |              +---------------+   |
|  |  load_artifact  |-------------+              | Returns       |   |
|  |  (filename,     |                            | Artifact      |   |
|  |   version?)     |-----------------+          | Content       |   |
|  +-----------------+                 |          +---------------+   |
|                                      |                              |
|  +-----------------+                 |          +---------------+   |
|  |  list_artifacts |                 +--------->| Storage       |   |
|  |  ()             |---------------------------->| Backend       |   |
|  +-----------------+                            +---------------+   |
|                                                                     |
+---------------------------------------------------------------------+
```

Load Latest Version

```python
async def load_report(context: CallbackContext):
    """Load latest version of report."""

    # Load most recent version
    artifact = await context.load_artifact('sales_report_q3_2025.md')

    if artifact:
        # Extract text content
        text = artifact.text
        print(f"Loaded report:\n{text}")
    else:
        print("Report not found")
```

Load Specific Version

```python
async def load_version(context: CallbackContext, filename: str, version: int):
    """Load specific artifact version."""

    # Load version 1 (second save) of the file
    # Remember: versions are 0-indexed (0=first, 1=second, 2=third)
    artifact = await context.load_artifact(
        filename=filename,
        version=version
    )

    if artifact:
        print(f"Loaded {filename} version {version}")
        return artifact.text
    else:
        print(f"Version {version} not found")
        return None
```

Handle Missing Artifacts

```python
async def safe_load(context: CallbackContext, filename: str):
    """Safely load artifact with error handling."""

    try:
        artifact = await context.load_artifact(filename)

        if artifact is None:
            print(f"Artifact {filename} does not exist")
            return None

        return artifact.text

    except Exception as e:
        print(f"Error loading {filename}: {e}")
        return None
```

4. Listing Artifacts

List All Artifacts

```python
async def show_all_artifacts(context: CallbackContext):
    """List all available artifacts."""

    artifacts = await context.list_artifacts()

    print("Available Artifacts:")
    print("="*60)

    for filename in artifacts:
        print(f"  - {filename}")

    print(f"\nTotal: {len(artifacts)} artifacts")
```

Filter Artifacts by Type

```python
async def list_by_extension(context: CallbackContext, extension: str):
    """List artifacts by file extension."""

    all_artifacts = await context.list_artifacts()

    filtered = [
        f for f in all_artifacts
        if f.endswith(extension)
    ]

    print(f"Artifacts with extension {extension}:")
    for f in filtered:
        print(f"  - {f}")

    return filtered
```

Built-in Artifact Loading Tool

ADK provides a built-in tool for automatically loading artifacts into LLM context:

```python
from google.adk.tools.load_artifacts_tool import load_artifacts_tool
from google.adk.agents import Agent

# Add to your agent's tools
agent = Agent(
    name='artifact_agent',
    model='gemini-2.0-flash',
    tools=[
        load_artifacts_tool,  # Built-in artifact loader
        # ... your other tools
    ]
)
```

What it does:

- Automatically lists available artifacts for the agent
- Loads artifact content when the LLM requests it
- Handles both session-scoped and user-scoped artifacts

- Provides artifact content in the conversation context

When to use:

- When you want the LLM to discover and use artifacts automatically
- For conversational access to stored files
- When building document Q&A or analysis agents

5. Real-World Example: Document Processor

Let's build a document processing pipeline with comprehensive artifact management.

```
Document Processing Pipeline:

+----------------------------------------------------------------------+
|                    DOCUMENT PROCESSING WORKFLOW                      |
+----------------------------------------------------------------------+
|                                                                      |
|   Input Document                                                     |
|         |                                                            |
|         v                                                            |
|   +-------------------+                                              |
|   | 1. Extract Text  |-----> Artifact: document_extracted.txt (v0)  |
|   +-------------------+                                              |
|         |                                                            |
|         v                                                            |
|   +-------------------+                                              |
|   | 2. Summarize     |-----> Artifact: document_summary.txt (v0)   |
|   +-------------------+                                              |
|         |                                                            |
|         v                                                            |
|   +-------------------+                                              |
|   | 3. Translate     |-----> Artifact: document_Spanish.txt (v0)   |
|   |    (Spanish)     |-----> Artifact: document_French.txt (v0)    |
|   +-------------------+                                              |
|         |                                                            |
|         v                                                            |
|   +-------------------+                                              |
|   | 4. Create Report |-----> Artifact: document_FINAL_REPORT.md    |
|   +-------------------+       (combines all artifacts)              |
|         |                                                            |
|         v                                                            |
|   Final Output: Complete report with all processing stages          |
|                                                                      |
+----------------------------------------------------------------------+
```

Complete Implementation

```python
"""
Document Processor with Artifact Management
Processes documents through multiple stages with versioning and audit trails.
"""

import asyncio
import os
from datetime import datetime
from typing import Dict, Optional
from google.adk.agents import Agent, Runner, Session
from google.adk.tools import FunctionTool
from google.adk.tools.tool_context import ToolContext
from google.genai import types

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project-id'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

class DocumentProcessor:
    """Document processing pipeline with artifact tracking."""

    def __init__(self):
        """Initialize document processor."""

        # Processing history
        self.processing_log: list[Dict] = []

        # Create processing tools

        async def extract_text(document: str, tool_context: ToolContext) -> st
            """Extract and clean text from document."""

            self._log_step('extract_text', document)

            # Simulated text extraction
            extracted = f"EXTRACTED TEXT FROM: {document}\n\n"
            extracted += "This is the cleaned and extracted content..."

            # Save extracted version
            part = types.Part.from_text(extracted)
            version = await tool_context.save_artifact(
                filename=f"{document}_extracted.txt",
                part=part
            )
            return f"Text extracted and saved as version {version}"

        async def summarize_document(document: str, tool_context: ToolContext)
            """Generate document summary."""

            self._log_step('summarize', document)

            # Load original
            artifact = await tool_context.load_artifact(f"{document}_extracted

            if not artifact:
                return "Error: Extracted text not found"

            # Generate summary
            summary = f"SUMMARY OF {document}\n\n"
            summary += "Key Points:\n"
            summary += "- Point 1: Important finding\n"
            summary += "- Point 2: Critical insight\n"
            summary += "- Point 3: Key recommendation\n"

            # Save summary
            part = types.Part.from_text(summary)
            version = await tool_context.save_artifact(
                filename=f"{document}_summary.txt",
                part=part
            )

            return f"Summary created as version {version}"

        async def translate_document(document: str, language: str,
                                    tool_context: ToolContext) -> str:
            """Translate document to target language."""

            self._log_step('translate', f"{document} to {language}")

            # Load extracted text
            artifact = await tool_context.load_artifact(f"{document}_extracted

            if not artifact:
                return "Error: Source document not found"

            # Simulated translation
            translated = f"TRANSLATED ({language}): {document}\n\n"
            translated += f"[Content translated to {language}]"

            # Save translation
            part = types.Part.from_text(translated)
```

```
            version = await tool_context.save_artifact(
                filename=f"{document}_{language}.txt",
                part=part
            )

            return f"Translation to {language} saved as version {version}"

        async def create_report(document: str, tool_context: ToolContext) -> s
            """Create comprehensive report from all artifacts."""

            self._log_step('create_report', document)

            # List all artifacts for this document
            all_artifacts = await tool_context.list_artifacts()
            doc_artifacts = [a for a in all_artifacts if a.startswith(document

            # Build report
            report = f"""
# Document Processing Report
Document: {document}
Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

## Processing Pipeline

"""

            # Load and include each artifact
            for artifact_name in doc_artifacts:
                artifact = await tool_context.load_artifact(artifact_name)
                if artifact:
                    report += f"\n### {artifact_name}\n\n"
                    report += f"```\n{artifact.text[:200]}...\n```\n"

            report += f"\n## Artifacts Created\n\n"
            report += f"Total artifacts: {len(doc_artifacts)}\n"
            for name in doc_artifacts:
                report += f"- {name}\n"

            # Save final report
            part = types.Part.from_text(report)
            version = await tool_context.save_artifact(
                filename=f"{document}_FINAL_REPORT.md",
                part=part
            )

            return f"Final report created as version {version}"
```

```
        async def list_all_documents(tool_context: ToolContext) -> str:
            """List all processed documents."""

            artifacts = await tool_context.list_artifacts()

            # Extract unique document names
            documents = set()
            for artifact in artifacts:
                # Remove suffixes like _extracted, _summary, etc.
                base = artifact.split('_')[0]
                documents.add(base)

            result = "Processed Documents:\n"
            for doc in sorted(documents):
                result += f"- {doc}\n"

            return result

        # Create document processor agent
        self.agent = Agent(
            model='gemini-2.0-flash',
            name='document_processor',
            description='Processes documents through multiple stages',
            instruction="""
You are a document processing agent with the following capabilities:

**Tools Available:**
1. extract_text: Extract and clean text from document
2. summarize_document: Generate document summary
3. translate_document: Translate to target language
4. create_report: Create comprehensive processing report
5. list_all_documents: List all processed documents

**Processing Workflow:**
For document processing requests:
1. Extract text first
2. Create summary
3. Translate if requested
4. Generate final report
5. Report all artifacts created

Always explain what you're doing at each step.
            """.strip(),
            tools=[
                FunctionTool(extract_text),
                FunctionTool(summarize_document),
                FunctionTool(translate_document),
                FunctionTool(create_report),
                FunctionTool(list_all_documents)
            ],
            generate_content_config=types.GenerateContentConfig(
                temperature=0.3,
                max_output_tokens=2048
            )
        )

        self.runner = Runner()
        self.session = Session()

    def _log_step(self, step: str, details: str):
        """Log processing step."""
        self.processing_log.append({
            'timestamp': datetime.now().isoformat(),
            'step': step,
            'details': details
        })

    async def process_document(self, document_name: str, operations: list[str]
        """
        Process document with specified operations.

        Args:
            document_name: Name of document to process
            operations: List of operations (extract, summarize, translate, rep
        """

        print(f"\n{'='*70}")
        print(f"PROCESSING: {document_name}")
        print(f"OPERATIONS: {', '.join(operations)}")
        print(f"{'='*70}\n")

        # Build query
        query = f"Process document '{document_name}' with operations: {', '.jo

        if 'translate' in operations:
            query += " (translate to Spanish and French)"

        # Execute processing
        result = await self.runner.run_async(
            query,
            agent=self.agent,
            session=self.session
        )
        print("\n📄 PROCESSING RESULT:\n")
        print(result.content.parts[0].text)
        print(f"\n{'='*70}\n")

    def get_processing_log(self) -> str:
        """Get processing log summary."""

        log = f"\nPROCESSING LOG\n{'='*70}\n"

        for entry in self.processing_log:
            log += f"\n[{entry['timestamp']}]\n"
            log += f"  Step: {entry['step']}\n"
            log += f"  Details: {entry['details']}\n"

        log += f"\n{'='*70}\n"
        log += f"Total Steps: {len(self.processing_log)}\n"

        return log

async def main():
    """Main entry point."""

    processor = DocumentProcessor()

    # Process document 1: Full pipeline
    await processor.process_document(
        document_name='contract_2025_Q3',
        operations=['extract', 'summarize', 'translate', 'report']
    )

    await asyncio.sleep(2)

    # Process document 2: Extract and summarize only
    await processor.process_document(
        document_name='technical_spec_v2',
        operations=['extract', 'summarize', 'report']
    )

    await asyncio.sleep(2)

    # List all documents
    result = await processor.runner.run_async(
        "List all processed documents and their artifacts",
        agent=processor.agent,
        session=processor.session
    )
    print("\n📊 ALL DOCUMENTS:\n")
    print(result.content.parts[0].text)

    # Show processing log
    print(processor.get_processing_log())

if __name__ == '__main__':
    asyncio.run(main())
```

Expected Output

```
======================================================================
PROCESSING: contract_2025_Q3
OPERATIONS: extract, summarize, translate, report
======================================================================

📄 PROCESSING RESULT:

I'll process the document 'contract_2025_Q3' through the complete pipeline:

**Step 1: Text Extraction**
Text extracted and saved as version 0

**Step 2: Summarization**
Summary created as version 0

**Step 3: Translation to Spanish**
Translation to Spanish saved as version 0

**Step 4: Translation to French**
Translation to French saved as version 0

**Step 5: Final Report**
Final report created as version 0

**Processing Complete!**

Artifacts created:
- contract_2025_Q3_extracted.txt (v0)
- contract_2025_Q3_summary.txt (v0)
- contract_2025_Q3_Spanish.txt (v0)
- contract_2025_Q3_French.txt (v0)
- contract_2025_Q3_FINAL_REPORT.md (v0)

All stages completed successfully. The document has been extracted, summarized
translated to Spanish and French, and a comprehensive report has been generate

======================================================================


======================================================================
PROCESSING: technical_spec_v2
OPERATIONS: extract, summarize, report
======================================================================

📄 PROCESSING RESULT:

Processing 'technical_spec_v2':
```

```
**Step 1: Text Extraction**
Text extracted and saved as version 0

**Step 2: Summarization**
Summary created as version 0

**Step 3: Final Report**
Final report created as version 0

**Artifacts created:**
- technical_spec_v2_extracted.txt (v0)
- technical_spec_v2_summary.txt (v0)
- technical_spec_v2_FINAL_REPORT.md (v0)

Processing complete.

======================================================================

📊 ALL DOCUMENTS:

Processed Documents:
- contract_2025_Q3
  * contract_2025_Q3_extracted.txt
  * contract_2025_Q3_summary.txt
  * contract_2025_Q3_Spanish.txt
  * contract_2025_Q3_French.txt
  * contract_2025_Q3_FINAL_REPORT.md

- technical_spec_v2
  * technical_spec_v2_extracted.txt
  * technical_spec_v2_summary.txt
  * technical_spec_v2_FINAL_REPORT.md

Total: 2 documents, 8 artifacts

PROCESSING LOG
======================================================================

[2025-10-08T14:30:15.123456]
  Step: extract_text
  Details: contract_2025_Q3

[2025-10-08T14:30:16.234567]
  Step: summarize
  Details: contract_2025_Q3
```

```
[2025-10-08T14:30:17.345678]
  Step: translate
  Details: contract_2025_Q3 to Spanish

[2025-10-08T14:30:18.456789]
  Step: translate
  Details: contract_2025_Q3 to French

[2025-10-08T14:30:19.567890]
  Step: create_report
  Details: contract_2025_Q3

[2025-10-08T14:30:22.678901]
  Step: extract_text
  Details: technical_spec_v2

[2025-10-08T14:30:23.789012]
  Step: summarize
  Details: technical_spec_v2

[2025-10-08T14:30:24.890123]
  Step: create_report
  Details: technical_spec_v2

======================================================================
Total Steps: 8
```

6. Credential Management

:::warning Advanced Topic
Credential management in ADK uses the authentication framework with `AuthConfig` objects. This is more complex than simple key-value storage. For most use cases, consider using **session state** for API keys instead.
:::

```
Credential Storage Options:

+----------------------------------------------------------------------+
|                     CREDENTIAL MANAGEMENT                            |
+----------------------------------------------------------------------+
|                                                                      |
|  Simple Approach (Recommended):                                     |
|  +---------------------------+     +--------------------------+      |
|  | Session State Storage     |---->| API Keys in State        |      |
|  | context.state['api_key']  |     | Easy to Use              |      |
|  +---------------------------+     +--------------------------+      |
|                                                                      |
|  Advanced Approach (Production):                                    |
|  +---------------------------+     +--------------------------+      |
|  | Authentication Framework  |---->| AuthConfig + Credential  |      |
|  | save_credential()         |     | OAuth, Tokens, etc.      |      |
|  | load_credential()         |     | Secure Storage           |      |
|  +---------------------------+     +--------------------------+      |
|                                                                      |
+----------------------------------------------------------------------+
```

Simple API Key Storage (Recommended)

For simple API key storage, use session state:

```python
from google.adk.agents import CallbackContext

async def store_api_key(context: CallbackContext, service: str, key: str):
    """Store API key in session state."""

    # Store in session state
    context.state[f'{service}_api_key'] = key
    print(f"API key for {service} stored in session")

async def get_api_key(context: CallbackContext, service: str) -> Optional[str]
    """Retrieve API key from session state."""

    # Load from session state
    key = context.state.get(f'{service}_api_key')

    if key:
        print(f"API key for {service} retrieved")
        return key
    else:
        print(f"No API key found for {service}")
        return None
```

Using API Keys in Tools

```python
from google.adk.tools import FunctionTool
from google.adk.tools.tool_context import ToolContext

async def call_external_api(query: str, tool_context: ToolContext) -> str:
    """Call external API using stored API key."""

    # Load API key from state
    api_key = tool_context.state.get('openai_api_key')

    if not api_key:
        return "Error: API key not configured"

    # Use API key for external call
    # response = requests.post(
    #     url,
    #     headers={'Authorization': f'Bearer {api_key}'}
    # )

    return "API call successful"
```

Advanced: Authentication Framework

For production credential management with OAuth, API tokens, and secure storage:

Official Credential API:

```python
from google.adk.agents import CallbackContext
from google.adk.auth.auth_credential import AuthCredential
from google.adk.tools import AuthConfig

async def save_credential_advanced(
    context: CallbackContext,
    auth_config: AuthConfig
):
    """Save credential using authentication framework."""
    await context.save_credential(auth_config)

async def load_credential_advanced(
    context: CallbackContext,
    auth_config: AuthConfig
) -> Optional[AuthCredential]:
    """Load credential using authentication framework."""
    return await context.load_credential(auth_config)
```

:::info Learn More
For complete authentication patterns including OAuth, API authentication, and secure credential storage, see:

- **Tutorial 15**: Authentication & Security (coming soon)
- **Official Docs**: Authentication Guide (https://google.github.io/adk-docs/tools/authentication/)

The credential API requires understanding `AuthConfig` construction and the authentication framework. For simple use cases, session state is sufficient.
:::

7. Best Practices

✔️ DO: Use Descriptive Filenames

```python
# ✔️ Good - Clear, descriptive names
await context.save_artifact('sales_report_2025_Q3.pdf', part)
await context.save_artifact('customer_data_export_2025_10_08.csv', part)
await context.save_artifact('product_image_SKU_12345.png', part)

# ❌ Bad - Unclear names
await context.save_artifact('report.pdf', part)
await context.save_artifact('data.csv', part)
await context.save_artifact('image.png', part)
```

✔️ DO: Handle Missing Artifacts

```python
# ✔️ Good - Check existence
artifact = await context.load_artifact('report.txt')

if artifact:
    process(artifact.text)
else:
    print("Artifact not found, creating new one")
    # Create new artifact

# ❌ Bad - No error handling
artifact = await context.load_artifact('report.txt')
process(artifact.text)  # Crashes if artifact is None
```

✔️ DO: Track Artifact Provenance

```
# ✔️ Good - Include metadata in content
report = f"""
# Sales Report

Generated by: {agent_name}
Date: {datetime.now().isoformat()}
Version: {version}
Source Data: orders_2025_Q3.csv

[Report content...]
"""

await context.save_artifact('report.md', types.Part.from_text(report))

# ❌ Bad - No provenance
report = "[Report content...]"
await context.save_artifact('report.md', types.Part.from_text(report))
```

✔️ DO: Use Versioning Strategically

```
# ✔️ Good - Save at meaningful checkpoints
v1 = await context.save_artifact('analysis.txt', draft_part)
# ... user review ...
v2 = await context.save_artifact('analysis.txt', revised_part)
# ... final review ...
v3 = await context.save_artifact('analysis.txt', final_part)

# Each version represents a significant state

# ❌ Bad - Over-versioning
for i in range(1000):
    await context.save_artifact('data.txt', part)  # 1000 versions!
```

✔️ DO: Clean Up Sensitive Data

```
# ✔️ Good - Don't store sensitive data in artifacts
sanitized_data = remove_pii(raw_data)
await context.save_artifact('data.csv', types.Part.from_text(sanitized_data))

# Store credentials separately
await context.save_credential('api_key', secret_key)

# ❌ Bad - Sensitive data in artifacts
await context.save_artifact('data.csv', types.Part.from_text(raw_data_with_pii
```

8. Advanced Patterns

```
Advanced Artifact Patterns:

+----------------------------------------------------------------------+
|                       ADVANCED PATTERNS                              |
+----------------------------------------------------------------------+
|                                                                      |
|  Pattern 1: Diff Tracking                                           |
|  +--------------------+     +--------------------+                   |
|  | Version N-1        |---->| Compare Versions   |                   |
|  +--------------------+     +--------------------+                   |
|  | Version N          |---->| Generate Diff      |                   |
|  +--------------------+     +--------------------+                   |
|                                                                      |
|  Pattern 2: Pipeline Processing                                     |
|  +-----------+     +-----------+     +-----------+     +-----------+    |
|  | Input     |---->| Stage 1   |---->| Stage 2   |---->| Output    |    |
|  | Artifact |      | Artifact |      | Artifact |      | Artifact |    |
|  +-----------+     +-----------+     +-----------+     +-----------+    |
|                                                                      |
|  Pattern 3: Metadata Embedding                                      |
|  +--------------------------------------------------------------+     |
|  | Artifact Content                                             |     |
|  | +----------------------------------------------------------+ |     |
|  | | Metadata: {author, timestamp, version, tags}           | |     |
|  | +----------------------------------------------------------+ |     |
|  | | Actual Content: {...}                                  | |     |
|  | +----------------------------------------------------------+ |     |
|  +--------------------------------------------------------------+     |
|                                                                      |
+----------------------------------------------------------------------+
```

Pattern 1: Artifact Diff Tracking

```python
async def track_changes(context: CallbackContext, filename: str):
    """Track changes between artifact versions."""

    # Load current and previous versions
    current = await context.load_artifact(filename)

    if not current:
        return "No artifact found"

    # Assuming current version is 3, load version 2
    current_version = 3  # In production, track this
    previous = await context.load_artifact(filename, version=current_version -

    if previous:
        # Compare versions
        changes = compare_text(previous.text, current.text)
        return f"Changes: {changes}"
    else:
        return "First version"
```

Pattern 2: Artifact Pipeline

```python
async def process_pipeline(context: CallbackContext, input_file: str):
    """Process file through multiple stages."""

    # Stage 1: Load input
    input_artifact = await context.load_artifact(input_file)

    # Stage 2: Transform
    transformed = transform(input_artifact.text)
    v1 = await context.save_artifact(f"{input_file}_transformed",
                                     types.Part.from_text(transformed))

    # Stage 3: Analyze
    analyzed = analyze(transformed)
    v2 = await context.save_artifact(f"{input_file}_analyzed",
                                     types.Part.from_text(analyzed))

    # Stage 4: Report
    report = generate_report(analyzed)
    v3 = await context.save_artifact(f"{input_file}_report",
                                     types.Part.from_text(report))

    return f"Pipeline complete: {v1}, {v2}, {v3}"
```

Pattern 3: Artifact Metadata

```python
import json

async def save_with_metadata(context: CallbackContext, filename: str,
                            content: str, metadata: dict):
    """Save artifact with embedded metadata."""

    # Embed metadata in content
    wrapped = {
        'metadata': metadata,
        'content': content
    }

    json_str = json.dumps(wrapped, indent=2)
    part = types.Part.from_text(json_str)

    version = await context.save_artifact(filename, part)

    return version

async def load_with_metadata(context: CallbackContext, filename: str):
    """Load artifact and extract metadata."""

    artifact = await context.load_artifact(filename)

    if not artifact:
        return None, None

    data = json.loads(artifact.text)

    return data['content'], data['metadata']
```

9. Troubleshooting

Issue: "Artifacts Tab is Empty" (UI Display Issue)

:::info Expected Behavior
This is the #1 most common "issue" - but it's not actually a problem!

The Artifacts tab appears empty when using `InMemoryArtifactService`, but your artifacts **ARE being saved correctly**. This is a UI display limitation, not a functionality issue.
:::

What's happening:

- ✔️ Artifacts are being saved (check server logs for HTTP 200 responses)
- ✔️ Artifacts are being retrieved correctly
- ✔️ REST API is working perfectly
- ❌ Artifacts sidebar doesn't populate (UI limitation only)

How to verify artifacts are working:

1. **Check server logs** - Look for successful saves: `INFO: GET .../artifacts/document_extracted.txt/versions/0 HTTP/1.1" 200 OK INFO: GET .../artifacts/document_summary.txt/versions/0 HTTP/1.1" 200 OK`
2. **Look for blue buttons in chat** - Agent creates buttons like "display document_extracted.txt"
3. These buttons work perfectly
4. Click them to view artifact content
5. This is the **primary way** to access artifacts in development
6. **Ask the agent** - Use conversational access: `"Show me all saved artifacts" "Load document_extracted.txt" "What artifacts have been created?"`

Why does this happen?

The ADK web UI's Artifacts sidebar expects specific metadata hooks that `InMemoryArtifactService` doesn't provide. The artifacts exist in memory and are fully functional via:

- ✔️ REST API endpoints (confirmed by logs)
- ✔️ Blue button displays (confirmed by UI)
- ✔️ Agent tool calls (confirmed by implementation)
- ✔️ Programmatic access (confirmed by tests)

Production deployment:

In production with `GcsArtifactService`, the Artifacts sidebar **will populate correctly** because the cloud backend provides the necessary metadata indexing.

```
from google.adk.artifacts import GcsArtifactService

# Production configuration - sidebar will work
artifact_service = GcsArtifactService(bucket_name='your-bucket')
```

:::tip Workaround Summary
1. **Primary**: Click blue artifact buttons in chat
2. **Secondary**: Ask agent "Show me all saved artifacts"
3. **Tertiary**: Check server logs for confirmation
4. **Production**: Use GcsArtifactService for full UI support
:::

Issue: "Artifact not found"

Solutions:

1. **Check filename spelling**:

```
# List all artifacts to verify name
artifacts = await context.list_artifacts()
print("Available:", artifacts)
```

1. **Verify artifact was saved**:

```python
# Check save return value
version = await context.save_artifact('file.txt', part)
if version is not None:
    print(f"Saved successfully as version {version}")
else:
    print("Save failed")
```

1. **Check session scope**:

```python
# Artifacts are scoped to sessions
# Make sure you're in the same session
print(f"Current session: {context.session.id}")
```

Issue: "Version conflict"

Solution: Always use returned version:

```python
# ✔️ Good
v1 = await context.save_artifact('file.txt', part1)
v2 = await context.save_artifact('file.txt', part2)
# v1 = 0, v2 = 1 (0-indexed versions)

# Load specific version
artifact = await context.load_artifact('file.txt', version=v1)
```

Issue: "TypeError: save_artifact() got unexpected keyword argument"

Solution: Use correct parameter names (changed in ADK 1.16.0+):

```
# ✔ Correct - use 'artifact' parameter
await tool_context.save_artifact(
    filename='document.txt',
    artifact=types.Part.from_text(text)
)

# ❌ Wrong - old 'part' parameter
await tool_context.save_artifact(
    filename='document.txt',
    part=types.Part.from_text(text)  # This will fail
)
```

Issue: "Artifact service not configured"

Solution: Ensure artifact service is passed to Runner:

```
from google.adk.artifacts import InMemoryArtifactService

# ✔ Good - artifact service configured
runner = Runner(
    agent=agent,
    artifact_service=InMemoryArtifactService()
)

# ❌ Bad - no artifact service
runner = Runner(agent=agent)  # Will fail when calling artifact methods
```

Summary

You've mastered artifacts and file management:

Key Takeaways:

- ✔ `save_artifact()` stores files with automatic versioning
- ✔ `load_artifact()` retrieves specific or latest versions
- ✔ `list_artifacts()` discovers all stored files

- ✔️ `save_credential()` and `load_credential()` for secrets
- ✔️ Available in `CallbackContext` and `ToolContext`
- ✔️ All versions retained for audit trails
- ✔️ Perfect for document pipelines and reports

Production Checklist:

- [ ] Descriptive, unique filenames used
- [ ] Error handling for missing artifacts
- [ ] Provenance metadata included
- [ ] Sensitive data handled separately (credentials)
- [ ] Version tracking strategy defined
- [ ] Artifact retention policy established
- [ ] Regular cleanup of obsolete versions
- [ ] Monitoring of artifact storage usage

Next Steps:

- **Tutorial 20**: Learn YAML Configuration
- **Tutorial 21**: Explore Multimodal & Image Generation
- **Tutorial 22**: Master Model Selection

Resources:

- ADK Artifacts Documentation (https://google.github.io/adk-docs/artifacts/)
- Callback Context API (https://google.github.io/adk-docs/api/callback-context/)
- Tool Context API (https://google.github.io/adk-docs/api/tool-context/)

🎉 **Tutorial 19 Complete!** You now know how to manage files and artifacts with versioning. Continue to Tutorial 20 to learn about YAML configuration.

Generated on 2025-10-19 17:56:56 from 19_artifacts_files.md

Source: Google ADK Training Hub