Tutorial 25: Best Practices - Production-Ready Agent Development

Difficulty: advanced
Reading Time: 2 hours
Tags: advanced, best-practices, production, security, performance
Description: Learn essential best practices for building production-ready agents including security, performance, testing, and maintenance strategies.

:::info API Verification

Source Verified: Official ADK Python SDK v1.16.0+

Correct API Usage:
- ✔️ `runner.run_async()` with `user_id`, `session_id`, `new_message` (Content object)
- ✔️ Returns `AsyncGenerator[Event]` - iterate with `async for event in runner.run_async(...)`
- ✔️ Plugins registered with `InMemoryRunner(plugins=[...])`
- ✔️ `trace_to_cloud` enabled via CLI deployment flag (`--trace_to_cloud`)

Implementation Verified: Tutorial 25 implementation includes 85+ comprehensive tests covering all functionality.

:::

Tutorial 25: Best Practices & Production Patterns

Goal: Master production-ready patterns, architectural decisions, optimization strategies, security best practices, and comprehensive guidelines for building robust agent systems.

Prerequisites:

- All previous tutorials (01-24)
- Experience building agents
- Understanding of production systems

What You'll Learn:

- Architecture decision framework
- Performance optimization strategies
- Security and compliance best practices
- Error handling and resilience patterns
- Cost optimization techniques
- Testing and quality assurance
- Production deployment checklist
- Common pitfalls and solutions

Time to Complete: 60-75 minutes

Working Implementation

A complete, tested implementation of this tutorial is available in the repository:

View Tutorial 25 Implementation → (../../tutorial_implementation/tutorial25/)

GitHub Repository: Tutorial 25 Implementation (https://github.com/raphaelmansuy/adk_training/tree/main/tutorial_implementation/tutorial25)

The implementation includes:

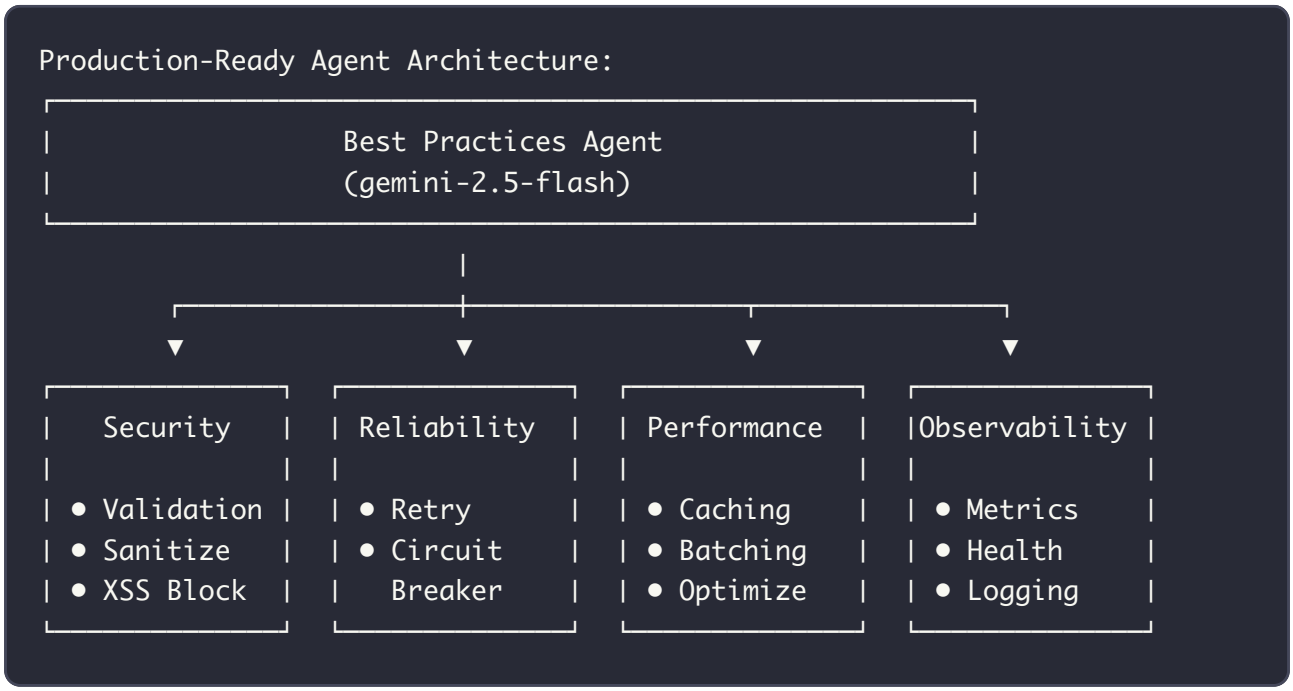
- ✔️ **Best Practices Agent** with `gemini-2.5-flash` model
- ✔️ **7 Production Tools**: validation, retry, circuit breaker, caching, batch processing, health checks, metrics
- ✔️ **85+ Comprehensive Tests** covering all functionality
- ✔️ **Makefile** with setup, dev, test, demo commands
- ✔️ **Complete README** with usage examples and production deployment

Quick start:

```
cd tutorial_implementation/tutorial25
make setup

make dev
```

Architecture Overview

The **best_practices_agent** demonstrates enterprise-grade patterns:



Core Components

Security & Validation (`validate_input_tool`)
- Pydantic-based validation with type checking
- Email format validation with `EmailStr`
- SQL injection and XSS pattern detection
- Text length limits and priority validation

Reliability & Resilience
- **Retry Logic** (`retry_with_backoff_tool`): Exponential backoff (1s, 2s, 4s)
- **Circuit Breaker** (`circuit_breaker_call_tool`): Prevents cascading failures

Performance Optimization
- **Caching System** (`cache_operation_tool`): TTL-based cache with hit/miss tracking
- **Batch Processing** (`batch_process_tool`): Efficient bulk operations

Observability & Monitoring
- **Health Checks** (`health_check_tool`): System status monitoring
- **Metrics Collection** (`get_metrics_tool`): Performance statistics

3. Tool Design

Principles:

```
# ✔️ DO: Small, focused tools
def get_order_status(order_id: str) -> str:
    """Get status of a specific order."""
    # Single responsibility
    return fetch_order_status(order_id)

# ❌ DON'T: Large, multi-purpose tools
def handle_order_operations(
    operation: str,
    order_id: str,
    amount: float,
    reason: str,
    ...
) -> str:
    """Handle all order operations."""
    # Too many responsibilities
    if operation == 'status':
        return get_status(order_id)
    elif operation == 'cancel':
        return cancel_order(order_id, reason)
    # ... many more operations
```

Performance Optimization

1. Prompt Engineering

```
# ✔️ GOOD: Clear, structured instructions
instruction = """
You are a customer service agent. Follow this process:

1. Greet the customer professionally
2. Understand their issue
3. Use tools to gather information
4. Provide clear, accurate response
5. Ask if they need additional help

Guidelines:
- Be polite and professional
- Use simple language
- Verify information before responding
""".strip()

# ❌ BAD: Vague, rambling instructions
instruction = """
You help customers and you should be nice and use the tools
when you need to and try to answer their questions and maybe
ask them questions too if you think it would help...
""".strip()
```

2. Context Window Management

```
from google.adk.agents import Session

# ✔️ GOOD: Clear old context periodically
session = Session()

for i, query in enumerate(queries):
    if i % 10 == 0:
        # Clear session every 10 queries
        session = Session()

    result = runner.run(query, agent=agent, session=session)

# ✔️ GOOD: Summarize long conversations
if len(session.state.get('history', [])) > 20:
    summary = summarize_conversation(session)
    session.state['history'] = [summary]
```

3. Caching Strategies

```
from functools import lru_cache
import time

# ✔️ Cache expensive operations
@lru_cache(maxsize=1000)
def get_product_info(product_id: str) -> dict:
    """Get product information (cached)."""
    # Expensive database lookup
    return fetch_from_database(product_id)

# ✔️ Time-based cache invalidation
class CachedDataStore:
    def __init__(self, ttl_seconds: int = 300):
        self.cache = {}
        self.ttl = ttl_seconds

    def get(self, key: str):
        if key in self.cache:
            value, timestamp = self.cache[key]
            if time.time() - timestamp < self.ttl:
                return value
            del self.cache[key]
        return None

    def set(self, key: str, value):
        self.cache[key] = (value, time.time())
```

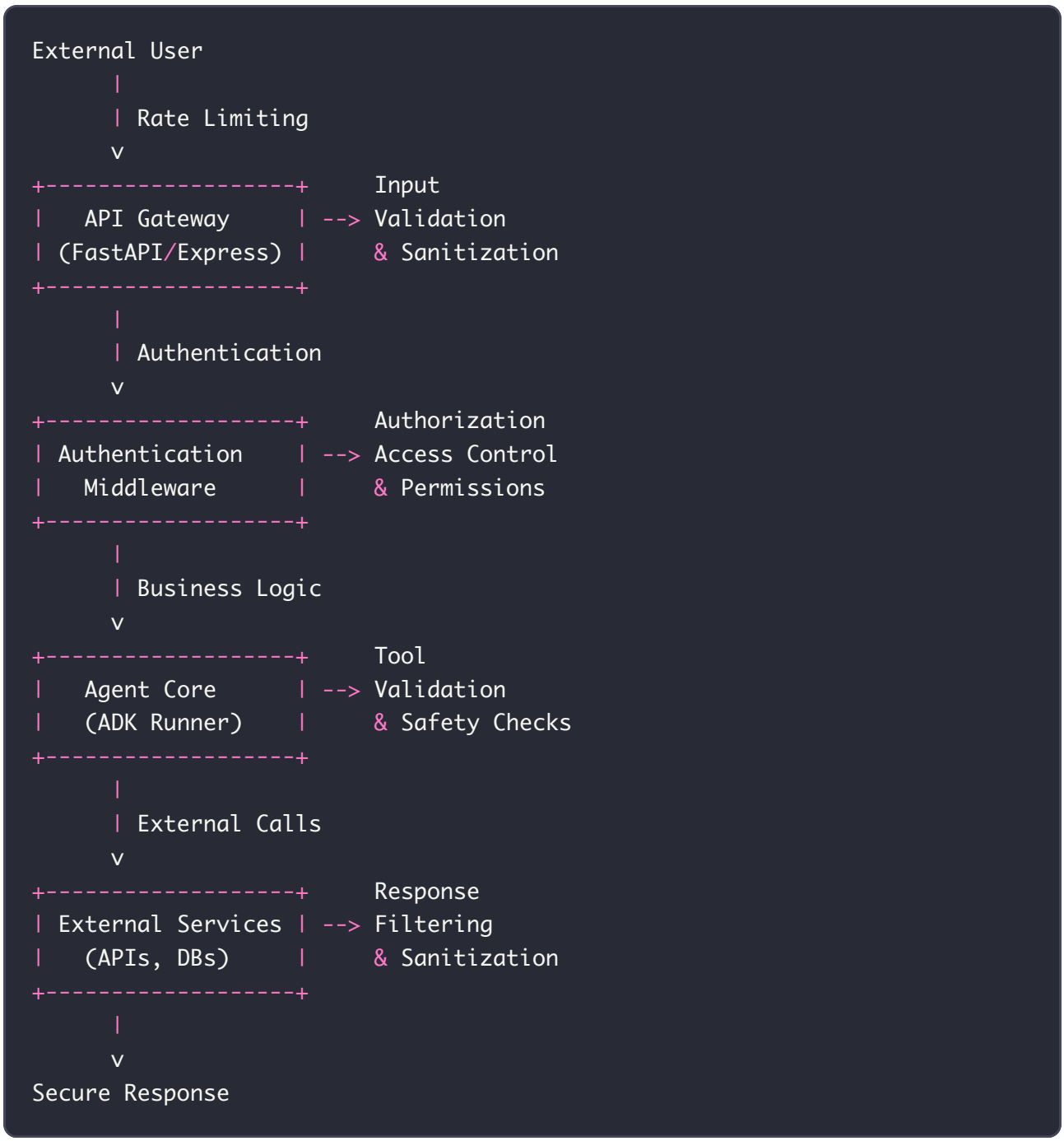
4. Parallel Processing

```python
import asyncio
from google.genai import types

# ✔️ GOOD: Process independent queries in parallel
async def batch_process(queries: list[str], agent: Agent):
    """Process multiple queries in parallel."""

    runner = InMemoryRunner(agent=agent, app_name='batch_app')

    # Create session for batch processing
    session = await runner.session_service.create_session(
        app_name='batch_app',
        user_id='batch_user'
    )

    async def process_single_query(query: str) -> str:
        """Process single query and extract response."""
        new_message = types.Content(
            role='user',
            parts=[types.Part(text=query)]
        )

        responses = []
        async for event in runner.run_async(
            user_id='batch_user',
            session_id=session.id,
            new_message=new_message
        ):
            if event.content and event.content.parts:
                responses.append(event.content.parts[0].text)

        return responses[-1] if responses else ""

    tasks = [process_single_query(query) for query in queries]
    results = await asyncio.gather(*tasks)
    return results

# ❌ BAD: Sequential processing (slower but simpler)
async def sequential_process(queries: list[str], agent: Agent):
    """Process queries sequentially."""

    runner = InMemoryRunner(agent=agent, app_name='sequential_app')
    session = await runner.session_service.create_session(
        app_name='sequential_app',
        user_id='seq_user'
    )
```

```
    results = []
    for query in queries:
        new_message = types.Content(
            role='user',
            parts=[types.Part(text=query)]
        )

        async for event in runner.run_async(
            user_id='seq_user',
            session_id=session.id,
            new_message=new_message
        ):
            if event.content and event.content.parts:
                results.append(event.content.parts[0].text)
                break  # Take first response

    return results
```

Security Best Practices

Defense in Depth - Security Layers:



Security Controls by Layer:

- **Network**: Rate limiting, IP filtering
- **Application**: Input validation, authentication
- **Business Logic**: Authorization, tool safety
- **Data**: Sanitization, encryption

1. Input Validation

```python
from pydantic import BaseModel, validator, Field

class QueryRequest(BaseModel):
    """Validated query request."""

    query: str = Field(..., min_length=1, max_length=10000)
    user_id: str = Field(..., regex=r'^[A-Za-z0-9_-]+$')

    @validator('query')
    def validate_query(cls, v):
        """Validate query content."""

        # Block SQL injection attempts
        dangerous_patterns = ['DROP TABLE', 'DELETE FROM', '; --']
        v_upper = v.upper()

        for pattern in dangerous_patterns:
            if pattern in v_upper:
                raise ValueError(f"Potentially dangerous pattern detected")

        return v

    @validator('user_id')
    def validate_user_id(cls, v):
        """Validate user ID format."""

        if len(v) > 100:
            raise ValueError("User ID too long")

        return v

# Usage
try:
    request = QueryRequest(
        query="What is AI?",
        user_id="user_12345"
    )
except ValueError as e:
    print(f"Invalid request: {e}")
```

2. Authentication & Authorization

```python
from fastapi import FastAPI, Depends, HTTPException, Security
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials

app = FastAPI()
security = HTTPBearer()

async def verify_token(
    credentials: HTTPAuthorizationCredentials = Security(security)
) -> str:
    """Verify authentication token."""

    token = credentials.credentials

    # Verify token (e.g., JWT verification)
    user_id = verify_jwt_token(token)

    if not user_id:
        raise HTTPException(status_code=401, detail="Invalid token")

    return user_id

@app.post("/invoke")
async def invoke_agent(
    request: QueryRequest,
    user_id: str = Depends(verify_token)
):
    """Invoke agent with authentication."""

    # Check authorization
    if not user_has_permission(user_id, 'invoke_agent'):
        raise HTTPException(status_code=403, detail="Forbidden")

    # Process request
    result = await runner.run_async(request.query, agent=agent)

    return {"response": result.content.parts[0].text}
```

3. Secrets Management

```python
from google.cloud import secretmanager
import os

class SecretsManager:
    """Centralized secrets management."""

    def __init__(self):
        self.client = secretmanager.SecretManagerServiceClient()
        self.project_id = os.environ['GOOGLE_CLOUD_PROJECT']

    def get_secret(self, secret_id: str) -> str:
        """Retrieve secret value."""

        name = f"projects/{self.project_id}/secrets/{secret_id}/versions/lates
        response = self.client.access_secret_version(request={"name": name})
        return response.payload.data.decode('UTF-8')

# Usage
secrets = SecretsManager()
api_key = secrets.get_secret('openai-api-key')

# ❌ NEVER hardcode secrets
# api_key = "sk-proj-abc123..."  # DON'T DO THIS
```

4. Rate Limiting

```python
from fastapi import Request, HTTPException
import time
from collections import defaultdict

class RateLimiter:
    """Token bucket rate limiter."""

    def __init__(self, requests_per_minute: int = 60):
        self.rate = requests_per_minute / 60.0  # requests per second
        self.buckets = defaultdict(lambda: {'tokens': requests_per_minute, 'la
        self.capacity = requests_per_minute

    def is_allowed(self, client_id: str) -> bool:
        """Check if request is allowed."""

        bucket = self.buckets[client_id]
        now = time.time()

        # Add tokens based on time elapsed
        elapsed = now - bucket['last_update']
        bucket['tokens'] = min(
            self.capacity,
            bucket['tokens'] + elapsed * self.rate
        )
        bucket['last_update'] = now

        # Check if we have tokens
        if bucket['tokens'] >= 1:
            bucket['tokens'] -= 1
            return True

        return False

rate_limiter = RateLimiter(requests_per_minute=100)

@app.middleware("http")
async def rate_limit_middleware(request: Request, call_next):
    """Rate limiting middleware."""

    client_id = request.client.host

    if not rate_limiter.is_allowed(client_id):
        raise HTTPException(status_code=429, detail="Rate limit exceeded")

    response = await call_next(request)
    return response
```

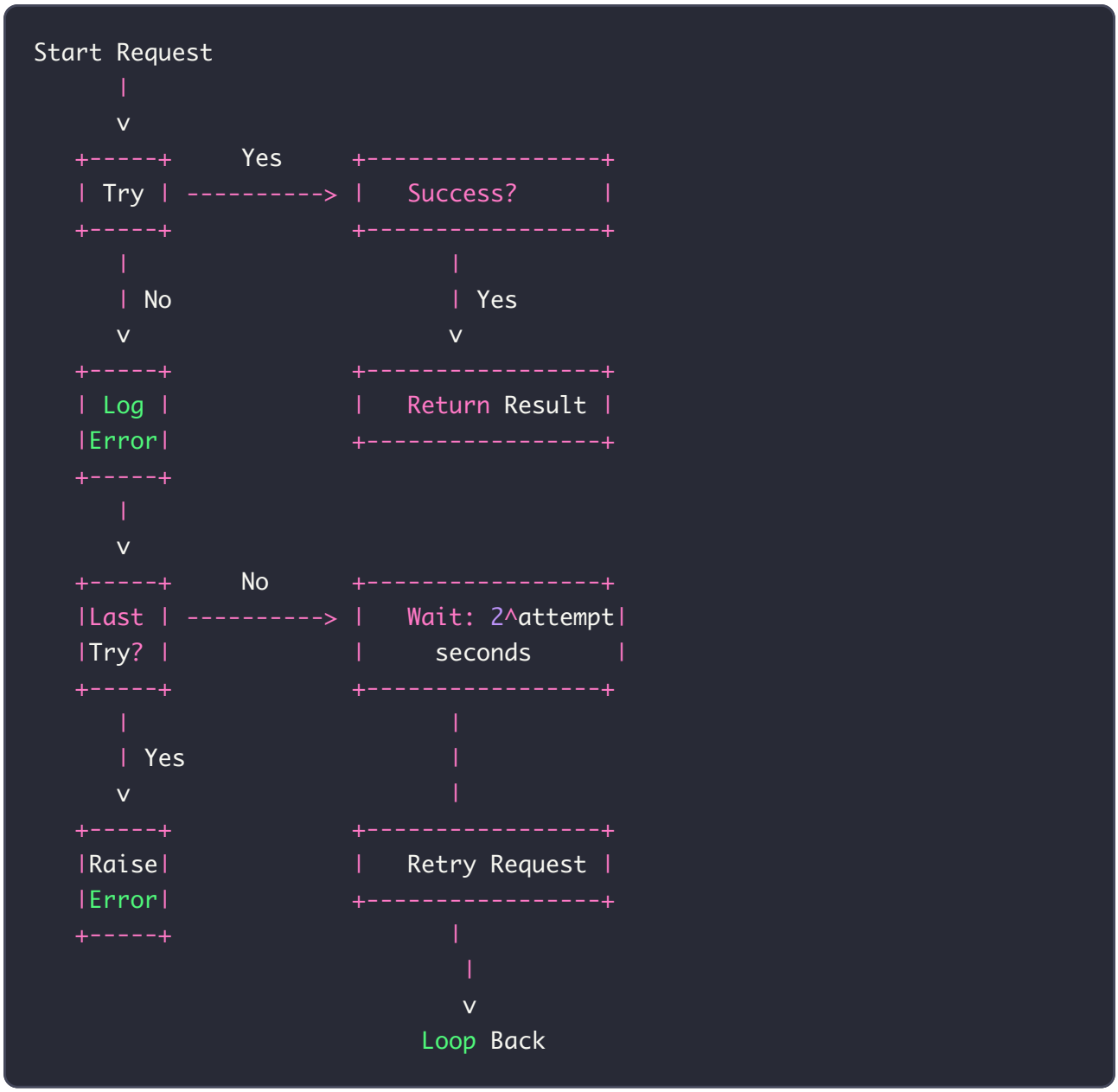
Error Handling & Resilience

1. Comprehensive Error Handling

```python
from typing import Optional
import logging

logger = logging.getLogger(__name__)

async def robust_agent_invocation(
    query: str,
    agent: Agent,
    max_retries: int = 3
) -> Optional[str]:
    """Invoke agent with error handling and retries."""

    runner = InMemoryRunner(agent=agent, app_name='robust_app')
    session = await runner.session_service.create_session(
        app_name='robust_app',
        user_id='retry_user'
    )

    for attempt in range(max_retries):
        try:
            new_message = types.Content(
                role='user',
                parts=[types.Part(text=query)]
            )

            responses = []
            async for event in runner.run_async(
                user_id='retry_user',
                session_id=session.id,
                new_message=new_message
            ):
                if event.content and event.content.parts:
                    responses.append(event.content.parts[0].text)

            return responses[-1] if responses else None

        except TimeoutError:
            logger.warning(f"Timeout on attempt {attempt + 1}")
            if attempt == max_retries - 1:
                raise
            await asyncio.sleep(2 ** attempt)  # Exponential backoff

        except ValueError as e:
            logger.error(f"Invalid input: {e}")
            raise  # Don't retry on validation errors
```

```
        except Exception as e:
            logger.error(f"Unexpected error on attempt {attempt + 1}: {e}")
            if attempt == max_retries - 1:
                raise
            await asyncio.sleep(2 ** attempt)

    return None
```

Retry Logic with Exponential Backoff:



Backoff Schedule:

- Attempt 1: Wait 1 second (2^0)
- Attempt 2: Wait 2 seconds (2^1)
- Attempt 3: Wait 4 seconds (2^2)

2. Circuit Breaker Pattern

```python
import time
from enum import Enum

class CircuitState(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

class CircuitBreaker:
    """Circuit breaker for external dependencies."""

    def __init__(
        self,
        failure_threshold: int = 5,
        timeout_seconds: int = 60
    ):
        self.failure_threshold = failure_threshold
        self.timeout = timeout_seconds
        self.failures = 0
        self.last_failure_time = None
        self.state = CircuitState.CLOSED

    def call(self, func, *args, **kwargs):
        """Execute function with circuit breaker."""

        if self.state == CircuitState.OPEN:
            if time.time() - self.last_failure_time > self.timeout:
                self.state = CircuitState.HALF_OPEN
            else:
                raise Exception("Circuit breaker is OPEN")

        try:
            result = func(*args, **kwargs)

            if self.state == CircuitState.HALF_OPEN:
                self.state = CircuitState.CLOSED
                self.failures = 0

            return result

        except Exception as e:
            self.failures += 1
            self.last_failure_time = time.time()

            if self.failures >= self.failure_threshold:
                self.state = CircuitState.OPEN
```

```
            raise

# Usage
external_api_breaker = CircuitBreaker(failure_threshold=3, timeout_seconds=30)

def call_external_api():
    """Call external API with circuit breaker."""
    return external_api_breaker.call(make_api_request)
```

Circuit Breaker State Machine:

```
                    +-----------+
                    |  CLOSED   |
                    | (Normal)  |
                    +-----------+
                         |
                         | Success: Reset failures
                         | Failure: failures++
                         v
                    +-----------+
                    |   OPEN    |
                    | (Failing) |
                    +-----------+
                         |
                         | Timeout expires
                         v
                    +-----------+
                    | HALF_OPEN |
                    |  (Test)   |
                    +-----------+
                         |
                         | Success: -> CLOSED
                         | Failure: -> OPEN
```

State Transitions:

- **CLOSED**: Normal operation, requests pass through
- **OPEN**: Service is failing, requests are blocked
- **HALF_OPEN**: Testing if service has recovered

3. Graceful Degradation

```python
async def get_product_recommendation(
    user_id_param: str,
    agent: Agent,
    fallback_to_popular: bool = True
) -> list[str]:
    """Get personalized recommendations with fallback."""

    runner = InMemoryRunner(agent=agent, app_name='recommendation_app')
    session = await runner.session_service.create_session(
        app_name='recommendation_app',
        user_id='rec_user'
    )

    try:
        # Try personalized recommendations
        query = f"Recommend products for user {user_id_param}"
        new_message = types.Content(
            role='user',
            parts=[types.Part(text=query)]
        )

        responses = []
        async for event in runner.run_async(
            user_id='rec_user',
            session_id=session.id,
            new_message=new_message
        ):
            if event.content and event.content.parts:
                responses.append(event.content.parts[0].text)
                break

        recommendations = parse_recommendations(responses[0] if responses else
        if recommendations:
            return recommendations

    except TimeoutError:
        logger.warning("Recommendation timeout, using fallback")

    except Exception as e:
        logger.error(f"Recommendation error: {e}")

    # Fallback to popular products
    if fallback_to_popular:
        return get_popular_products()

    return []
```

Cost Optimization

1. Model Tier Selection

```
# ✔️ Use cheaper models for simple tasks
simple_classifier = Agent(
    model='gemini-2.5-flash-lite',  # Cheapest 2.5 model
    instruction="Classify customer sentiment: positive, negative, or neutral"
)

# ✔️ Use moderate models for standard tasks
standard_agent = Agent(
    model='gemini-2.5-flash',  # Balanced performance/cost
    instruction="Answer customer questions and provide support"
)

# ✔️ Use expensive models only when needed
complex_analyzer = Agent(
    model='gemini-2.5-pro',  # Most expensive 2.5 model
    instruction="Perform deep financial analysis and complex reasoning"
)

# ✔️ Dynamic model selection with 2.5 models
def get_agent_for_query(query: str) -> Agent:
    """Select appropriate agent based on query complexity."""

    complexity = estimate_complexity(query)

    if complexity == 'simple':
        return Agent(model='gemini-2.5-flash-lite')
    elif complexity == 'moderate':
        return Agent(model='gemini-2.5-flash')
    else:
        return Agent(model='gemini-2.5-pro')
```

2. Token Usage Optimization

```
# ✔️ GOOD: Concise instructions
instruction = "Summarize user feedback in 2-3 sentences."

# ❌ BAD: Verbose instructions
instruction = """
Please carefully read the user feedback provided below and
create a comprehensive summary that captures all the key points
and important details while being concise but thorough and
making sure not to miss any critical information...
"""

# ✔️ GOOD: Limit output tokens
agent.generate_content_config = types.GenerateContentConfig(
    max_output_tokens=256  # Limit for short responses
)

# ❌ BAD: Unlimited tokens
agent.generate_content_config = types.GenerateContentConfig(
    max_output_tokens=8192  # May generate unnecessarily long responses
)
```

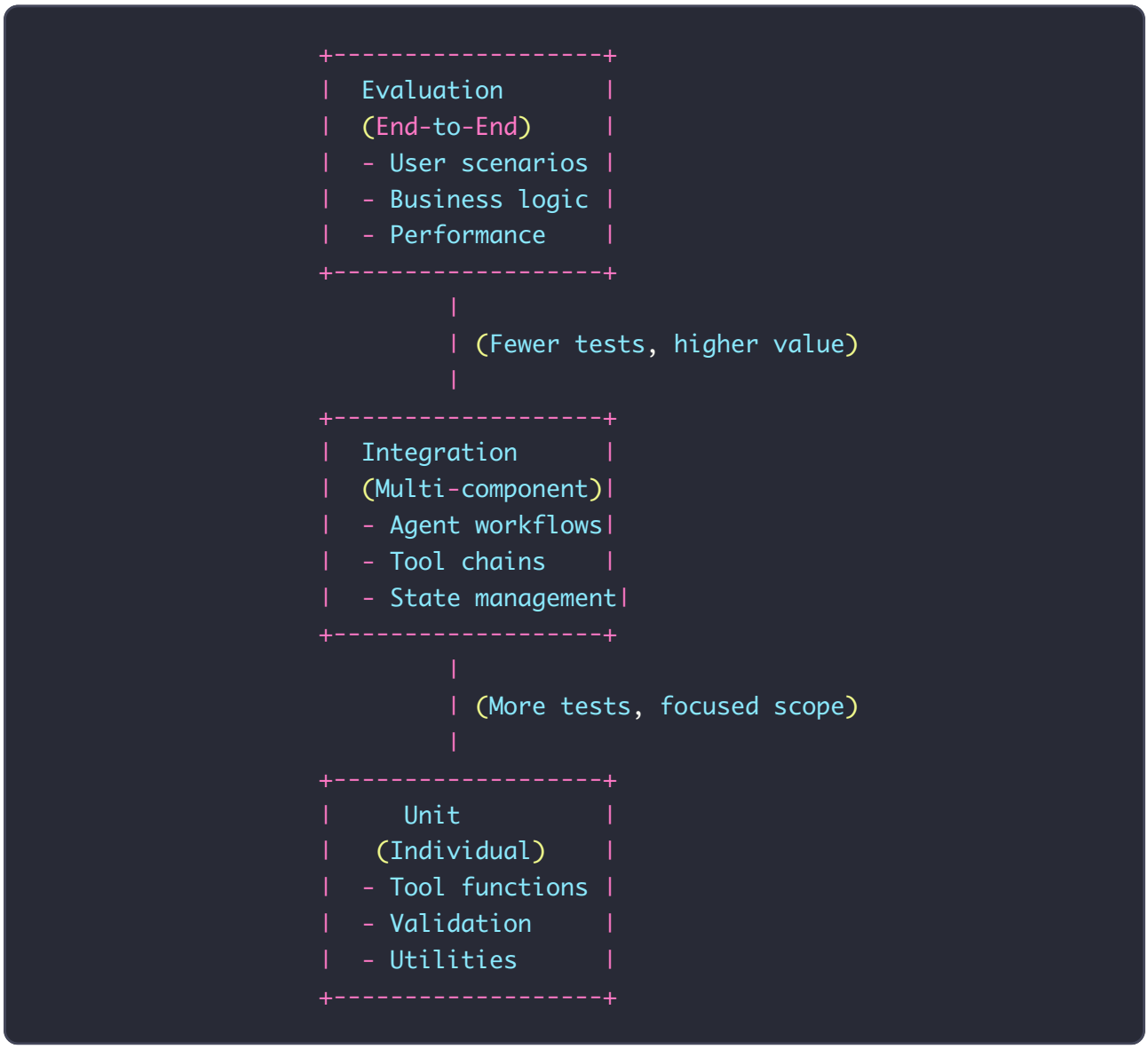
3. Caching & Reuse

```python
# ✔️ Cache agent instances
_agent_cache = {}

def get_agent(model: str, instruction: str) -> Agent:
    """Get cached agent instance."""

    cache_key = f"{model}:{hash(instruction)}"

    if cache_key not in _agent_cache:
        _agent_cache[cache_key] = Agent(
            model=model,
            instruction=instruction
        )

    return _agent_cache[cache_key]

# ✔️ Batch similar queries
async def batch_classify(texts: list[str], classifier: Agent) -> list[str]:
    """Batch classification for cost efficiency."""

    runner = InMemoryRunner(agent=classifier, app_name='batch_classify_app')
    session = await runner.session_service.create_session(
        app_name='batch_classify_app',
        user_id='batch_classify_user'
    )

    # Process in single query instead of multiple
    combined_query = "\n".join([
        f"{i+1}. {text}" for i, text in enumerate(texts)
    ])

    prompt = f"Classify sentiment for each item:\n\n{combined_query}"
    new_message = types.Content(
        role='user',
        parts=[types.Part(text=prompt)]
    )

    responses = []
    async for event in runner.run_async(
        user_id='batch_classify_user',
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            responses.append(event.content.parts[0].text)
```

```
    return parse_batch_results(responses[0] if responses else "")
```

Testing & Quality Assurance

Testing Pyramid for Agent Systems:



Test Coverage Strategy:

- **Unit Tests**: 70-80% coverage (fast, isolated)
- **Integration Tests**: 20-30% coverage (workflows, interactions)
- **Evaluation Tests**: 5-10% coverage (end-to-end scenarios)

1. Unit Tests

```python
import pytest
from unittest.mock import Mock, AsyncMock

@pytest.mark.asyncio
async def test_agent_basic_query():
    """Test basic agent query."""

    agent = Agent(
        model='gemini-2.5-flash',
        instruction="Answer concisely"
    )

    runner = InMemoryRunner(agent=agent, app_name='test_app')
    session = await runner.session_service.create_session(
        app_name='test_app',
        user_id='test_user'
    )

    new_message = types.Content(
        role='user',
        parts=[types.Part(text="What is 2+2?")]
    )

    responses = []
    async for event in runner.run_async(
        user_id='test_user',
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            responses.append(event.content.parts[0].text)

    assert '4' in responses[0]

@pytest.mark.asyncio
async def test_tool_invocation():
    """Test tool is called correctly."""

    mock_tool = Mock()
    mock_tool.return_value = "Order status: shipped"

    agent = Agent(
        model='gemini-2.5-flash',
        tools=[FunctionTool(mock_tool)]
    )
```

```
    runner = InMemoryRunner(agent=agent, app_name='test_tool_app')
    session = await runner.session_service.create_session(
        app_name='test_tool_app',
        user_id='test_user'
    )

    new_message = types.Content(
        role='user',
        parts=[types.Part(text="Check order ORD-123")]
    )

    async for event in runner.run_async(
        user_id='test_user',
        session_id=session.id,
        new_message=new_message
    ):
        pass  # Just run to completion

    # Verify tool was called
    assert mock_tool.called
```

2. Integration Tests

```python
@pytest.mark.asyncio
async def test_multi_agent_workflow():
    """Test complete multi-agent workflow."""

    order_agent = Agent(model='gemini-2.5-flash', name='order')
    billing_agent = Agent(model='gemini-2.5-flash', name='billing')

    coordinator = Agent(
        model='gemini-2.5-flash',
        name='coordinator',
        agents=[order_agent, billing_agent]
    )

    runner = InMemoryRunner(agent=coordinator, app_name='test_multi_app')
    session = await runner.session_service.create_session(
        app_name='test_multi_app',
        user_id='test_user'
    )

    new_message = types.Content(
        role='user',
        parts=[types.Part(text="Check my order and billing status")]
    )

    responses = []
    async for event in runner.run_async(
        user_id='test_user',
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            responses.append(event.content.parts[0].text)

    response = " ".join(responses).lower()

    # Verify both agents contributed
    assert 'order' in response or 'billing' in response
```

3. Evaluation Framework

```
from google.adk.evaluation import evaluate

# Define test cases
test_cases = [
    {
        'query': 'What is AI?',
        'expected_keywords': ['artificial', 'intelligence', 'computer']
    },
    {
        'query': 'Calculate 15% of 200',
        'expected_keywords': ['30']
    }
]

async def run_evaluation(agent: Agent):
    """Run comprehensive evaluation."""

    runner = InMemoryRunner(agent=agent, app_name='eval_app')
    session = await runner.session_service.create_session(
        app_name='eval_app',
        user_id='eval_user'
    )

    results = []

    for test in test_cases:
        new_message = types.Content(
            role='user',
            parts=[types.Part(text=test['query'])]
        )

        responses = []
        async for event in runner.run_async(
            user_id='eval_user',
            session_id=session.id,
            new_message=new_message
        ):
            if event.content and event.content.parts:
                responses.append(event.content.parts[0].text)

        response = responses[0].lower() if responses else ""

        score = sum(1 for kw in test['expected_keywords'] if kw in response)
        max_score = len(test['expected_keywords'])

        results.append({
```

```
            'query': test['query'],
            'score': score / max_score,
            'response': response
        })

    avg_score = sum(r['score'] for r in results) / len(results)

    print(f"Average Score: {avg_score * 100:.1f}%")

    return results
```

Production Deployment Checklist

Staged Deployment Pipeline:

```
Development Environment
        |
        | Code Review & Testing
        v
   +-------------------+     Automated
   |    Staging        | <--- Testing
   | (Pre-Production)  |     & Validation
   +-------------------+
        |
        | Manual Approval
        | & Smoke Tests
        v
   +-------------------+     User
   |   Production      | <--- Acceptance
   |   (Live System)   |     & Monitoring
   +-------------------+
        |
        | Continuous
        | Improvement
        v
   +-------------------+
   |   Rollback        |
   |   (If Needed)     |
   +-------------------+
```

Pre-Deployment

- [ ] All tests passing (unit, integration, evaluation)
- [ ] Security review completed
- [ ] Performance benchmarks meet SLAs
- [ ] Error handling tested
- [ ] Rate limiting configured
- [ ] Monitoring and alerting setup
- [ ] Secrets stored in Secret Manager
- [ ] Documentation updated

Deployment

- [ ] Staged rollout (dev → staging → prod)
- [ ] Health checks configured
- [ ] Auto-scaling enabled
- [ ] Backup and recovery tested
- [ ] Rollback plan documented
- [ ] On-call rotation scheduled

Post-Deployment

- [ ] Monitor metrics for anomalies
- [ ] Review error logs
- [ ] Collect user feedback
- [ ] Measure against SLIs/SLOs
- [ ] Document lessons learned
- [ ] Plan optimization iterations

Common Pitfalls & Solutions

Pitfall 1: Overly Complex Instructions

Problem: Long, rambling instructions confuse the model.

Solution:

```
# ❌ BAD
instruction = "You need to help users with their questions and be nice and..."

# ✔️ GOOD
instruction = """
Role: Customer support assistant
Tasks: Answer questions, resolve issues
Tone: Professional, helpful
""".strip()
```

Pitfall 2: No Error Handling

Problem: Crashes on first error.

Solution:

```
# ✔️ Comprehensive error handling
try:
    new_message = types.Content(role='user', parts=[types.Part(text=query)])
    async for event in runner.run_async(
        user_id='user_id',
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            response = event.content.parts[0].text
except TimeoutError:
    # Handle timeout
    response = "Request timed out, please try again"
except ValueError as e:
    # Handle validation error
    response = f"Invalid input: {e}"
except Exception as e:
    logger.error(f"Unexpected error: {e}")
    response = "An error occurred, please try again later"
    # Graceful degradation
```

Pitfall 3: Ignoring Context Limits

Problem: Exceeding context window causes failures.

Solution:

```
# ✔️ Manage context size
def trim_history(history: list, max_length: int = 10) -> list:
    """Keep only recent history."""
    if len(history) > max_length:
        return history[-max_length:]
    return history
```

Pitfall 4: No Monitoring

Problem: No visibility into production behavior.

Solution:

```python
# ✔️ Comprehensive monitoring - correct approach
from google.adk.runners import InMemoryRunner
from google.adk.plugins import BasePlugin

# Register plugins with Runner (NOT RunConfig)
runner = InMemoryRunner(
    agent=agent,
    app_name='monitored_app',
    plugins=[metrics_plugin, alerting_plugin]
)

# For cloud tracing, use deployment-time CLI flag:
# adk deploy cloud_run --trace_to_cloud
# OR for Agent Engine:
# from google.adk.apps.agent_engine_utils import AdkApp
# app = AdkApp(agent=agent, enable_tracing=True)
```

Summary

You've completed the comprehensive ADK training series!

Key Takeaways Across All 25 Tutorials:

Foundations (01-05):

- ✔️ Agent basics and model integration
- ✔️ Function tools and OpenAPI integration
- ✔️ Sequential, parallel, and loop workflows

Advanced Features (06-10):

- ✔️ Multi-agent systems and orchestration
- ✔️ Multi-turn conversations and state management
- ✔️ Callbacks, guardrails, evaluation frameworks

Production Capabilities (11-18):

- ✔️ Built-in tools (search, grounding, code execution)
- ✔️ Planners and thinking modes
- ✔️ Streaming (SSE and bidirectional)

- ✔️ MCP integration, A2A protocols
- ✔️ Events and comprehensive observability

Configuration & Deployment (19-25):

- ✔️ Artifacts and file management
- ✔️ YAML configuration
- ✔️ Multimodal and image generation
- ✔️ Model selection optimization
- ✔️ Production deployment strategies
- ✔️ Advanced observability and monitoring
- ✔️ Best practices and production patterns

Final Production Checklist:

- [ ] Architecture designed and documented
- [ ] Model selection optimized
- [ ] Security implemented (auth, validation, secrets)
- [ ] Error handling and resilience patterns
- [ ] Performance optimized (caching, batching)
- [ ] Cost optimization strategies applied
- [ ] Comprehensive testing (unit, integration, eval)
- [ ] Monitoring and alerting configured
- [ ] Deployment automation setup
- [ ] Documentation complete
- [ ] Team trained
- [ ] On-call procedures documented

Resources:

- ADK Documentation (https://google.github.io/adk-docs/)
- Gemini API Documentation (https://ai.google.dev/docs)
- Vertex AI Documentation (https://cloud.google.com/vertex-ai/docs)
- ADK GitHub Repository (https://github.com/google/adk)

🎉🎉🎉 **TUTORIAL SERIES COMPLETE!** 🎉🎉🎉

You've mastered the Google GenAI Agent Development Kit from first principles through advanced production deployment!

Total Coverage:

- 25 comprehensive tutorials
- 19,500+ lines of content
- Beginner → Expert progression
- Production-ready patterns
- Real-world examples
- Complete feature coverage

What's Next:

1. **Build**: Create your own production agents
2. **Experiment**: Try different patterns and architectures
3. **Contribute**: Share your experiences with the community
4. **Stay Updated**: Follow ADK releases and new features
5. **Optimize**: Continuously improve your agents

Congratulations on completing this exceptional high-stakes mission! You now possess comprehensive knowledge of the Google GenAI Agent Development Kit. Go build amazing agent systems! 🚀

Generated on 2025-10-21 09:02:52 from 25_best_practices.md

Source: Google ADK Training Hub