

pyWATS

Python API for WATS Test Data Management

Official Documentation

Generated: 2026-01-26

Table of Contents

Introduction

Overview

Getting Started

Getting Started

Quick Reference

Installation

Overview

Api

Client

Gui

Docker

Windows Service

Linux Service

Macos Service

Architecture

Architecture

Client Architecture

Integration Patterns

Domain Modules

Overview

Report

Asset

Process

Product

Production

Analytics

Rootcause

Software

Scim

Events

Usage Guides

[Report Module](#)

[Report Builder](#)

[Asset Module](#)

[Process Module](#)

[Product Module](#)

[Production Module](#)

[Rootcause Module](#)

[Software Module](#)

[Box Build Guide](#)

Reference

[Env Variables](#)

[Error Catalog](#)

[Wats Domain Knowledge](#)

[Llm Converter Guide](#)

Introduction

Source: `docs/README.md`

Documentation Folder Structure

This folder contains **official, user-facing documentation** that ships with `pip install pywats-api`.

Published Documentation (in this folder)

Getting Started

- **getting-started.md** - Complete installation, configuration, logging, and error handling guide
- **INDEX.md** - Documentation index and navigation

Installation Guides

Choose by component and use case:

- **installation/** - Installation overview with decision tree
- **installation/api.md** - Python SDK only
- **installation/client.md** - Client service with queue
- **installation/gui.md** - Desktop GUI application
- **installation/docker.md** - Container deployment
- **installation/windows-service.md** - Windows service
- **installation/linux-service.md** - Linux systemd
- **installation/macos-service.md** - macOS launchd

Domain API Documentation

These files are included in the PyPI package:

- **modules/product.md** - Product domain API reference
- **modules/asset.md** - Asset domain API reference

- **modules/production.md** - Production domain API reference
- **modules/report.md** - Report domain API reference
- **modules/analytics.md** - Analytics domain API reference
- **modules/software.md** - Software domain API reference
- **modules/rootcause.md** - RootCause domain API reference
- **modules/process.md** - Process domain API reference

Module Usage Guides

Detailed guides with comprehensive examples:

- **usage/** - Detailed module guides (report-module.md, product-module.md, etc.)
- Detailed usage patterns
- Advanced examples
- Factory method documentation

Documentation Examples

Code snippets and examples embedded in documentation:

- **examples/** - Example code referenced in documentation
- `basic_usage.py` - Getting started example

Internal Documentation (NOT published)

All internal documentation is in separate folders:

- **internal_documentation/** - Architecture, design docs, AI agent knowledge, internal guides

These folders are excluded from the pip package.

Folder Structure

```
docs/
├── INDEX.md           ✓ Published - Documentation index
├── README.md          ✓ Published - This file
├── getting-started.md ✓ Published - Getting started guide
├── architecture.md    ✓ Published - System architecture
├── error-catalog.md   ✓ Published - Error reference
├── modules/
│   ├── product.md
│   ├── asset.md
│   ├── report.md
│   └── ...
├── usage/              ✓ Published - Detailed module guides
│   ├── report-module.md
│   ├── product-module.md
│   └── ...
└── ...
├── installation/      ✓ Published - Installation guides
│   ├── client.md
│   ├── docker.md
│   └── ...
├── internal_documentation/ ✗ NOT Published - Internal docs
│   ├── archived/
│   ├── WIP/
│   └── ...
└── domain_health/     ✗ NOT Published - Health tracking
```

Rule of Thumb

- **Files/folders in docs/ root** → Published with pip package
- **Folders: usage/ , modules/ , installation/** → Published (user-facing)
- **Folders: internal_documentation/ , domain_health/** → NOT Published (GitHub only)

Moving Documents

When creating new documentation:

- **User-facing API docs** → Put in docs/modules/
- **Detailed usage guides** → Put in docs/usage/
- **Installation guides** → Put in docs/installation/
- **Internal architecture/design** → Put in docs/internal_documentation/



Packaging

Controlled by `MANIFEST.in` in the project root:

- **Includes:** `docs/*.md` , `docs/usage/` , `docs/modules/` , `docs/installation/`
- **Excludes:** `docs/internal_documentation/` , `docs/domain_health/`

Getting Started

Source: [docs/getting-started.md](#)

Getting Started Guide

Complete guide to installing, configuring, and initializing pyWATS.

Table of Contents

- Installation
- API Initialization
- Async Usage
- Authentication
- Logging Configuration
- Exception Handling
- Performance Optimization
- Internal API Usage
- Client Installation
- Batch Operations & Pagination

Installation

Library Only (API Access)

For Python scripts and applications using the WATS API:

```
pip install pywats-api
```

With GUI Client

For desktop applications with Qt-based GUI (Windows, macOS, Linux):

```
pip install pywats-api[client]
```

Headless Client

For servers, Raspberry Pi, and embedded systems (no Qt/GUI):

```
pip install pywats-api[client-headless]
```

Development Installation

From source for development:

```
git clone https://github.com/olreppe/pyWATS.git
cd pyWATS
python -m venv .venv
source .venv/bin/activate # Linux/Mac
.venv\Scripts\activate # Windows
pip install -e ".[dev]"
```

API Initialization

Basic Initialization

```
from pywats import pyWATS

# Initialize with credentials
api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your_base64_token" # Base64 of "username:password"
)

# Test connection
if api.test_connection():
    print(f"Connected! Server version: {api.get_version()}")
```

Using Environment Variables

```
import os
from pywats import pyWATS

# Set environment variables
os.environ['WATS_BASE_URL'] = 'https://your-wats-server.com'
os.environ['WATS_AUTH_TOKEN'] = 'your_base64_token'

# Initialize from environment
api = pyWATS() # Automatically reads from environment
```

Or create a `.env` file:

```
WATS_BASE_URL=https://your-wats-server.com
WATS_AUTH_TOKEN=your_base64_token
```

```
from dotenv import load_dotenv
from pywats import pyWATS

load_dotenv() # Load from .env file
api = pyWATS()
```

Configuration Options

```
from pywats import pyWATS, RetryConfig
from pywats.core.exceptions import ErrorMode

api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your_token",

    # Optional: Custom timeout (default: 30 seconds)
    timeout=60,

    # Optional: Process cache refresh interval (default: 300 seconds)
    process_refresh_interval=600,

    # Optional: Error handling mode (default: STRICT)
    error_mode=ErrorMode.STRICT, # or ErrorMode.LENIENT

    # Optional: Retry configuration (default: enabled with 3 attempts)
    retry_enabled=True, # Set to False to disable
    # Or for advanced configuration:
    # retry_config=RetryConfig(max_attempts=5, base_delay=2.0)
)
```

Automatic Retry

The library automatically retries requests that fail due to transient errors:

- **Connection errors** - Network unavailable, DNS failures
- **Timeout errors** - Server took too long to respond
- **HTTP 429** - Too Many Requests (respects `Retry-After` header)
- **HTTP 500/502/503/504** - Server errors, often transient during deployments

Retry uses **exponential backoff with jitter** to avoid thundering herd:

Attempt	Delay (approx)
1	0-1 second
2	0-2 seconds
3	0-4 seconds
(fail)	Exception raised

Important: Only idempotent methods (GET, PUT, DELETE) are retried. POST is never retried automatically to prevent duplicate creates.

```
from pywats import pyWATS, RetryConfig

# Disable retry entirely
api = pyWATS(
    base_url="https://...",
    token="...",
    retry_enabled=False
)

# Custom retry configuration
config = RetryConfig(
    max_attempts=5,          # Try up to 5 times
    base_delay=2.0,           # Start with 2 second delay
    max_delay=60.0,           # Cap delay at 60 seconds
    jitter=True,              # Add randomness to avoid thundering herd
)
api = pyWATS(
    base_url="https://...",
    token="...",
    retry_config=config
)

# Check retry statistics
print(api.retry_config.stats)
# {'total_retries': 3, 'total_retry_time': 4.5}
```

Error Handling Modes

The library supports two error handling modes that control how missing data and 404 responses are handled:

STRICT Mode (Default - Recommended for Production)

```
from pywats import pyWATS
from pywats.core.exceptions import ErrorMode

api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your_token",
    error_mode=ErrorMode.STRICT # Default, can be omitted
)
```

Behavior:

- **404 errors** → Raises `NotFoundError`
- **Empty responses** (200 with no data) → Raises `EmptyResponseError`
- **All 4xx/5xx errors** → Raises appropriate exception
- **If no exception** → You have valid data (guaranteed)

Use STRICT when:

- Writing production code that needs certainty
- Validating data existence
- Critical operations where failures must be explicit
- You want type safety (no `None` checks needed)

Example:

```
from pywats import NotFoundError

try:
    product = api.product.get_product("WIDGET-001")
    # If we reach here, product is DEFINITELY valid
    print(f"Found: {product.part_number}")
except NotFoundError:
    print("Product doesn't exist - handle explicitly")
```

LENIENT Mode (Recommended for Scripts/Exploration)

```
from pywats import pyWATS
from pywats.core.exceptions import ErrorMode

api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your_token",
    error_mode=ErrorMode.LENIENT
)
```

Behavior:

- **404 errors** → Returns `None` (no exception)
- **Empty responses** → Returns `None` (no exception)
- **Actual errors** (5xx, 400, 401, 403, 409) → Still raises exceptions
- **Must check for `None`** before using returned data

Use LENIENT when:

- Writing exploratory scripts
- Querying optional data
- Batch processing where some items may not exist
- You want simpler code with less try/except boilerplate

Example:

```
# Simpler code - no try/except needed for missing data
product = api.product.get_product("WIDGET-001")

if product is None:
    print("Product doesn't exist")
else:
    print(f"Found: {product.part_number}")
```

Mode Comparison

Aspect	STRICT Mode	LENIENT Mode
404 Response	Raises <code>NotFoundError</code>	Returns <code>None</code>
Empty Response	Raises <code>EmptyResponseError</code>	Returns <code>None</code>
Server Error (5xx)	Raises <code>ServerError</code>	Raises <code>ServerError</code>
Validation Error (400)	Raises <code>ValidationError</code>	Raises <code>ValidationError</code>
Auth Error (401)	Raises <code>AuthenticationError</code>	Raises <code>AuthenticationError</code>
Permission Error (403)	Raises <code>AuthorizationError</code>	Raises <code>AuthorizationError</code>
Conflict (409)	Raises <code>ConflictError</code>	Raises <code>ConflictError</code>
Return Type	<code>Model</code> or raises	<code>Model</code> <code>None</code>
None Checks	Not needed	Required
Best For	Production code	Scripts/exploration

Choosing the Right Mode

```
# Production code - use STRICT for explicit error handling
from pywats.core.exceptions import ErrorMode

api_prod = pyWATS(..., error_mode=ErrorMode.STRICT)

try:
    product = api_prod.product.get_product(part_number)
    # Guaranteed to have valid product here
    process_product(product)
except NotFoundError as e:
    log_error(f"Product not found: {e}")
    send_alert(...)
except ValidationError as e:
    log_error(f"Invalid data: {e}")

# Exploratory script - use LENIENT for simpler code
api_explore = pyWATS(..., error_mode=ErrorMode.LENIENT)

for part_number in candidate_parts:
    product = api_explore.product.get_product(part_number)
    if product: # Simple None check
        print(f"Found: {product.part_number}")
    # Missing products are silently skipped
```

Station Configuration

Configure test station identity:

```
from pywats import pyWATS, Station, StationConfig, Purpose

# Configure station
station = Station(
    config=StationConfig(
        station_name="ICT-01",
        location="Production Line A"
    )
)

# Initialize with station
api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your_token",
    station=station
)

# Station is automatically used in reports
from pywats.tools.test_uut import TestUUT

uut = TestUUT(
    part_number="WIDGET-001",
    serial_number="SN-12345",
    revision="A",
    operator="John Doe",
    purpose=Purpose.TEST # or 10
)
# Station info automatically filled from api.station
```

Async Usage

pyWATS supports both synchronous and asynchronous usage patterns. The library is designed with an **async-first architecture** where all business logic lives in async services, and sync services are thin wrappers.

Synchronous Usage (Default)

The standard synchronous API is the easiest way to use pyWATS:

```
from pywats import pyWATS

api = pyWATS(base_url="https://...", token="...")

# Synchronous calls - simple and blocking
products = api.product.get_products()
unit = api.production.get_unit("SN-12345", "WIDGET-001")
```

Asynchronous Usage

For high-performance applications, use the `async` API directly:

```
import asyncio
from pywats import AsyncWATS

async def main():
    # Create async client
    async with AsyncWATS(base_url="https://...", token="...") as api:
        # Async calls - non-blocking
        products = await api.product.get_products()
        unit = await api.production.get_unit("SN-12345", "WIDGET-001")

        # Concurrent requests
        product, unit, assets = await asyncio.gather(
            api.product.get_product("WIDGET-001"),
            api.production.get_unit("SN-12345", "WIDGET-001"),
            api.asset.get_assets(top=10)
        )

    asyncio.run(main())
```

Using `run_sync()` for Mixed Code

When you have `async` code but need to call it from sync context:

```
from pywats.core.sync_runner import run_sync
from pywats import AsyncWATS

async def fetch_data():
    async with AsyncWATS(base_url="https://...", token="...") as api:
        return await api.product.get_products()

# Call async code from sync context
products = run_sync(fetch_data())
```

Service Architecture

All domains follow the same pattern:

Component	Description
AsyncXxxService	Source of truth - all business logic
XxxService	Thin sync wrapper using <code>run_sync()</code>
AsyncXxxRepository	Async data access layer

```
# Both use the same underlying logic
from pywats.domains.product.service import ProductService          # Sync
from pywats.domains.product.async_service import AsyncProductService # Async
```

Authentication

Token Generation

pyWATS uses Base64-encoded credentials:

```
import base64

username = "your_username"
password = "your_password"

# Create token
credentials = f"{username}:{password}"
token = base64.b64encode(credentials.encode()).decode()

print(f"Token: {token}")
```

Using in API

```
from pywats import pyWATS

api = pyWATS(
    base_url="https://your-wats-server.com",
    token=token # Use generated token
)
```

Token Security

Best Practices:

1. **Never hardcode credentials** in source code
2. **Use environment variables** for production
3. **Use .env files** for development (add to .gitignore)
4. **Rotate tokens** regularly
5. **Use separate tokens** for different environments

```
# ❌ DON'T DO THIS
api = pyWATS(base_url="https://...", token="dXNlcjpwYXNz")

# ✅ DO THIS
import os
api = pyWATS(
    base_url=os.getenv('WATS_BASE_URL'),
    token=os.getenv('WATS_AUTH_TOKEN')
)
```

Logging Configuration

Quick Debug Mode

Enable detailed logging for troubleshooting:

```
from pywats import pyWATS, enable_debug_logging

# Enable debug logging before creating API instance
enable_debug_logging()

api = pyWATS(base_url="...", token="...")
```

This shows:

- HTTP requests and responses
- API calls to WATS server
- Data serialization/deserialization
- Repository and service operations

Custom Logging Configuration

```
import logging
from pywats import pyWATS

# Configure logging your way
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('pywats.log'),
        logging.StreamHandler()
    ]
)

# Set pyWATS logger to DEBUG
logging.getLogger('pywats').setLevel(logging.DEBUG)

# Or be more specific
logging.getLogger('pywats.http_client').setLevel(logging.DEBUG)
logging.getLogger('pywats.domains.product').setLevel(logging.INFO)

api = pyWATS(base_url="...", token="...")
```

Log Levels

Level	Use Case	Output
DEBUG	Development, troubleshooting	All details including HTTP requests
INFO	Normal operation	Important operations and status
WARNING	Production monitoring	Warnings and potential issues
ERROR	Minimal logging	Errors only

Example Log Output

```
2026-01-08 14:30:15 - pywats.http_client - INFO - GET https://wats.example.com/api/Product/1234
2026-01-08 14:30:15 - pywats.http_client - DEBUG - Response: 200 OK (1234 bytes)
2026-01-08 14:30:15 - pywats.domains.product - DEBUG - Retrieved product: WIDGET-001
```

Disable Logging

```
import logging

# Disable all pyWATS logging
logging.getLogger('pywats').setLevel(logging.CRITICAL)
```

Exception Handling

Exception Hierarchy

pyWATS provides a comprehensive exception hierarchy for precise error handling:

```
PyWATSError
├── AuthenticationError      # Base exception for all pyWATS errors
├── AuthorizationError      # Authentication failed (401)
├── NotFoundError           # Permission denied (403)
├── ValidationError         # Resource not found (404)
├── ValidationDataError     # Invalid request data (400)
├── ConflictError           # Resource conflict (409)
├── EmptyResponseError       # Empty response in STRICT mode (200 with no data)
├── ServerError              # Server-side error (5xx)
├── ConnectionError          # Network/connection failure
└── TimeoutError             # Request timeout
```

All exceptions include:

- **message**: Human-readable error description
- **operation**: Name of the operation that failed (e.g., "get_product")
- **details**: Additional context without HTTP internals
- **cause**: Original exception if wrapping another error

Basic Error Handling

```
from pywats import pyWATS, PyWATSError, AuthenticationError

try:
    api = pyWATS(base_url="https://...", token="...")
    product = api.product.get_product("WIDGET-001")

except AuthenticationError:
    print("Invalid credentials - check your token")

except PyWATSError as e:
    print(f"WATS API error: {e}")
    print(f"Operation: {e.operation}")
    print(f"Details: {e.details}")

except Exception as e:
    print(f"Unexpected error: {e}")
```

Comprehensive Exception Handling

```
from pywats import (
    pyWATS,
    AuthenticationError,
    AuthorizationError,
    NotFoundError,
    ValidationError,
    ConflictError,
    EmptyResponseError,
    ServerError,
    ConnectionError,
    TimeoutError,
    PyWATSError
)

try:
    api = pyWATS(base_url="https://...", token="...")
    product = api.product.get_product("WIDGET-001")

except AuthenticationError as e:
    # 401 - Invalid credentials
    print(f"Authentication failed: {e.message}")
    # Action: Verify token, re-authenticate

except AuthorizationError as e:
    # 403 - No permission
    print(f"Permission denied: {e.message}")
    # Action: Check user permissions, request access

except NotFoundError as e:
    # 404 - Resource doesn't exist (STRICT mode only)
    print(f"Not found: {e.resource_type} '{e.identifier}'")
    # Action: Verify identifier, create resource

except ValidationError as e:
    # 400 - Invalid request data
    print(f"Validation error: {e.message}")
    if e.field:
        print(f"  Field: {e.field}")
    if e.value:
        print(f"  Value: {e.value}")
    # Action: Fix input data, check API documentation

except ConflictError as e:
    # 409 - Resource conflict
    print(f"Conflict: {e.message}")
    # Action: Resolve conflict, retry with updated data

except EmptyResponseError as e:
    # 200 with empty body (STRICT mode only)
    print(f"Empty response: {e.message}")
    # Action: Verify query parameters, switch to LENIENT mode

except ServerError as e:
```

```

# 5xx - Server-side error
print(f"Server error: {e.message}")
if e.status_code:
    print(f" Status: {e.status_code}")
# Action: Retry, contact support if persistent

except ConnectionError as e:
    # Network failure
    print(f"Connection failed: {e.message}")
    # Action: Check network, verify URL

except TimeoutError as e:
    # Request timeout
    print(f"Request timed out: {e.message}")
    # Action: Increase timeout, check network

except PyWATSError as e:
    # Catch-all for other errors
    print(f"WATS error: {e.message}")
    print(f"Operation: {e.operation}")
    print(f"Details: {e.details}")

```

Error Mode Impact on Exceptions

The `error_mode` parameter affects which exceptions are raised:

```

from pywats import pyWATS, NotFoundError, EmptyResponseError
from pywats.core.exceptions import ErrorMode

# STRICT mode - raises exceptions for missing data
api_strict = pyWATS(..., error_mode=ErrorMode.STRICT)

try:
    product = api_strict.product.get_product("UNKNOWN")
except NotFoundError:
    # This WILL be raised in STRICT mode
    print("Product not found")

try:
    product = api_strict.product.get_product("EMPTY-RESULT")
except EmptyResponseError:
    # This WILL be raised in STRICT mode
    print("Query returned no data")

# LENIENT mode - returns None for missing data
api_lenient = pyWATS(..., error_mode=ErrorMode.LENIENT)

product = api_lenient.product.get_product("UNKNOWN")
# Returns None - no NotFoundError raised

if product is None:
    print("Product not found or empty")

```

Exception Matrix by Mode:

Error Type	STRICT Mode	LENIENT Mode
404 Not Found	Raises <code>NotFoundError</code>	Returns <code>None</code>
Empty Response	Raises <code>EmptyResponseError</code>	Returns <code>None</code>
400 Validation	Raises <code>ValidationError</code>	Raises <code>ValidationError</code>
401 Auth	Raises <code>AuthenticationError</code>	Raises <code>AuthenticationError</code>
403 Permission	Raises <code>AuthorizationError</code>	Raises <code>AuthorizationError</code>
409 Conflict	Raises <code>ConflictError</code>	Raises <code>ConflictError</code>
5xx Server Error	Raises <code>ServerError</code>	Raises <code>ServerError</code>
Network Failure	Raises <code>ConnectionError</code>	Raises <code>ConnectionError</code>
Timeout	Raises <code>TimeoutError</code>	Raises <code>TimeoutError</code>

```

### Retry Logic

```python
import time
from pywats import pyWATS, ConnectionError, ServerError

def get_product_with_retry(api, part_number, max_retries=3):
 """Get product with automatic retry on network errors"""

 for attempt in range(max_retries):
 try:
 return api.product.get_product(part_number)

 except (ConnectionError, ServerError) as e:
 if attempt < max_retries - 1:
 wait_time = 2 ** attempt # Exponential backoff
 print(f'Retry {attempt + 1}/{max_retries} after {wait_time}s...')
 time.sleep(wait_time)
 else:
 raise # Re-raise on final attempt

 return None

Use it
try:
 product = get_product_with_retry(api, "WIDGET-001")
except PyWATSError as e:
 print(f'Failed after retries: {e}')

```

## Validation Errors

```

from pywats import ValidationError
from pywats.tools.test_uut import TestUUT

try:
 # Invalid data
 uut = TestUUT(
 part_number="", # Empty part number - will raise ValidationError
 serial_number="SN-12345",
 revision="A"
)

except ValidationError as e:
 print(f'Validation failed: {e}')
 # Handle gracefully - prompt user, use defaults, etc.

```

# Client Installation

---

## GUI Client (Desktop)

### Prerequisites

- Python 3.10 or later
- Qt6 support (Windows, macOS, Linux with display)

### Installation

```
Install with client support
pip install pywats-api[client]

Launch GUI
python -m pywats_client
```

Or from command line:

```
pywats-client
```

### First-Time Setup

1. Launch the client
2. Go to **Setup** tab
3. Configure:
4. Server URL: <https://your-wats-server.com>
5. API Token: Your Base64 token
6. Station Name: YOUR-STATION-01
7. Click **Save**
8. Click **Test Connection**

See **GUI Configuration Guide** for detailed setup.

### Using the File Menu

The GUI includes helpful controls in the **File** menu:

#### **Restart GUI** (Ctrl+R)

- Restarts only the GUI application
- Service continues running in the background
- Configuration is reloaded automatically
- Useful after GUI code changes during development

### **Stop Service** (Ctrl+Shift+S)

- Sends stop command to the service process
- Stops all file watching and converter workers
- GUI remains open but shows [Disconnected] status
- Useful before making service code changes

### **Exit** (Alt+F4)

- Closes the GUI application
- Service continues running if started separately

### **Window Title Status:**

- Shows [Connected] when service is running
  - Shows [Disconnected] when service is not reachable
- 

## **Headless Client (No GUI)**

For servers, Raspberry Pi, and embedded systems.

### **Installation**

```
Install headless client (no Qt)
pip install pywats-api[client-headless]
```

### **Initialize Configuration**

```
Interactive setup
pywats-client config init

Or non-interactive
pywats-client config init \
 --server-url https://wats.example.com \
 --api-token YOUR_TOKEN \
 --station-name RASPBERRY-PI-01 \
 --non-interactive
```

### **Test Connection**

```
pywats-client test-connection
```

## Start Service

```
Foreground (for testing)
pywats-client start

With HTTP API for remote management
pywats-client start --api --api-port 8765

As daemon (Linux/Unix)
pywats-client start --daemon
```

## Configuration File

Location:

- **Windows:** %APPDATA%\pyWATS\_Client\config.json
- **Linux/Mac:** ~/.config/pywats\_client/config.json

Example:

```
{
 "service_address": "https://wats.example.com",
 "api_token": "your_token",
 "station_name": "RASPBERRY-PI-01",
 "log_level": "INFO"
}
```

## CLI Commands

```
Configuration
pywats-client config show
pywats-client config get station_name
pywats-client config set log_level DEBUG

Service control
pywats-client status
pywats-client start
pywats-client stop

Converters
pywats-client converters list
pywats-client converters enable my_converter
```

See [Headless Operation Guide](#) for complete documentation.

## Complete Example

---

Putting it all together:

```

"""
Complete example of pyWATS initialization with logging and error handling
"""

import os
import logging
from dotenv import load_dotenv
from pywats import (
 pyWATS,
 enable_debug_logging,
 PyWATSError,
 AuthenticationError,
 Station,
 StationConfig,
 Purpose
)

def initialize_api():
 """Initialize pyWATS API with proper configuration"""

 # Load environment variables
 load_dotenv()

 # Enable debug logging if in debug mode
 if os.getenv('DEBUG', 'false').lower() == 'true':
 enable_debug_logging()
 else:
 # Configure INFO level logging
 logging.basicConfig(level=logging.INFO)

 # Get credentials from environment
 base_url = os.getenv('WATS_BASE_URL')
 token = os.getenv('WATS_AUTH_TOKEN')

 if not base_url or not token:
 raise ValueError("WATS_BASE_URL and WATS_AUTH_TOKEN must be set")

 # Configure station
 station = Station(
 config=StationConfig(
 station_name=os.getenv('WATS_STATION_NAME', 'DEFAULT-STATION'),
 location=os.getenv('WATS_LOCATION', 'Default Location')
)
)

 # Initialize API
 try:
 api = pyWATS(
 base_url=base_url,
 token=token,
 station=station,
 timeout=int(os.getenv('WATS_TIMEOUT', '30'))
)

 # Test connection

```

```
if api.test_connection():
 version = api.get_version()
 logging.info(f"Connected to WATS server version: {version}")
 return api
else:
 raise ConnectionError("Connection test failed")

except AuthenticationError:
 logging.error("Authentication failed - check credentials")
 raise

except PyWATSError as e:
 logging.error(f"Failed to initialize WATS API: {e}")
 raise

Use it
if __name__ == "__main__":
 try:
 api = initialize_api()

 # Now use the API
 products = api.product.get_products()
 print(f"Found {len(products)} products")

 except Exception as e:
 logging.exception("Application error")
 exit(1)
```

## Performance Optimization

---

pyWATS includes several performance optimizations to make your applications faster and more efficient:

### Enhanced TTL Caching

Cache static data automatically with TTL (Time To Live) expiration:

```

from pywats import AsyncWATS
from pywats.core.cache import AsyncTTLCache

async def main():
 async with AsyncWATS(base_url="https://...", token="...") as api:
 # Process service has built-in caching
 # First call - fetches from server
 processes = await api.process.get_processes()

 # Second call - returns cached data (100x faster!)
 processes = await api.process.get_processes()

 # Check cache statistics
 stats = api.process.cache_stats
 print(f"Cache hit rate: {stats['hit_rate']:.1%}") # e.g., 95.0%

 # Clear cache when needed
 await api.process.clear_cache()

Create your own caches for any data
cache = AsyncTTLCache[str](
 max_size=1000,
 default_ttl=300.0 # 5 minutes
)

async with cache:
 # Cache a value
 await cache.set("key", "value", ttl=60.0) # Custom TTL

 # Get a value
 value = await cache.get("key")

 # Check statistics
 print(f"Hit rate: {cache.stats.hit_rate:.1%}")
 print(f"Hits: {cache.stats.hits}, Misses: {cache.stats.misses}")

```

**Performance Impact:** 95% reduction in server calls for static data, 100x faster cache hits vs server calls.

## Connection Pooling

HTTP/2 connection pooling is automatically enabled for all API calls:

```
from pywats import AsyncWATS

async with AsyncWATS(base_url="https://...", token="...") as api:
 # All requests automatically use connection pooling
 # - Reuses connections (faster, less overhead)
 # - HTTP/2 multiplexing (multiple requests on one connection)
 # - Up to 100 max connections
 # - 20 keepalive connections

 # Concurrent requests are much faster
 import asyncio
 products, assets, units = await asyncio.gather(
 api.product.get_products(),
 api.asset.get_assets(top=100),
 api.production.get_units(top=100)
)
```

**Performance Impact:** 3-5x faster for bulk operations, automatic connection reuse.

## Request Batching

Process multiple items efficiently with built-in batching utilities:

```

from pywats import AsyncWATS
from pywats.core.batching import ChunkedBatcher, batch_map

async def main():
 async with AsyncWATS(base_url="https://...", token="...") as api:
 serial_numbers = [f"SN-{i:05d}" for i in range(1000)]

 # Method 1: ChunkedBatcher for size-based batching
 async with ChunkedBatcher(
 processor=lambda sns: api.production.get_units_batch(sns),
 chunk_size=50, # Process 50 at a time
 max_concurrent=5 # Up to 5 concurrent batches
) as batcher:
 units = await batcher.process(serial_numbers)

 # Method 2: batch_map for simple concurrent mapping
 async def fetch_unit(sn: str):
 return await api.production.get_unit(sn, "WIDGET-001")

 units = await batch_map(
 items=serial_numbers,
 func=fetch_unit,
 batch_size=50,
 max_concurrent=10
)

 print(f"Fetched {len(units)} units")

import asyncio
asyncio.run(main())

```

**Performance Impact:** 5-10x faster for bulk operations, automatic concurrency control.

## MessagePack Serialization

Use MessagePack for faster, smaller payloads (optional):

```

First, install MessagePack support
pip install msgpack

from pywats.core.performance import Serializer

Create serializer with MessagePack
serializer = Serializer(format='msgpack')

Serialize data
data = {"part_number": "WIDGET-001", "values": [1, 2, 3]}
payload = serializer.serialize(data)

Deserialize
result = serializer.deserialize(payload)

Compare formats
json_serializer = Serializer(format='json')
msgpack_serializer = Serializer(format='msgpack')

json_size = len(json_serializer.serialize(data))
msgpack_size = len(msgpack_serializer.serialize(data))

print(f"JSON: {json_size} bytes")
print(f"MessagePack: {msgpack_size} bytes ({msgpack_size/json_size:.1%} size)")
Output: MessagePack: 50% size of JSON

Benchmark serialization speed
from pywats.core.performance import benchmark_serialization

results = benchmark_serialization(data, iterations=10000)
for fmt, metrics in results.items():
 print(f"{fmt}: {metrics['ops_per_sec']:.0f} ops/sec")
MessagePack is typically 3x faster than JSON

```

**Performance Impact:** 50% smaller payloads, 3x faster serialization, graceful fallback to JSON if not installed.

## Combined Performance Pattern

Use all optimizations together for maximum performance:

```

from pywats import AsyncWATS
from pywats.core.cache import AsyncTTLCache
from pywats.core.batching import batch_map

async def process_production_data():
 async with AsyncWATS(base_url="https://...", token="...") as api:
 # 1. Use built-in caching for static data
 processes = await api.process.get_processes() # Cached automatically

 # 2. Create custom cache for frequently accessed data
 product_cache = AsyncTTLCache[dict](max_size=1000, default_ttl=600.0)

 async with product_cache:
 # 3. Batch process units with connection pooling
 serial_numbers = [f"SN-{i:05d}" for i in range(1000)]

 async def fetch_with_cache(sn: str):
 # Check cache first
 cached = await product_cache.get(sn)
 if cached:
 return cached

 # Fetch from API (uses connection pooling automatically)
 unit = await api.production.get_unit(sn, "WIDGET-001")

 # Cache result
 await product_cache.set(sn, unit)
 return unit

 # Process in batches with concurrency control
 units = await batch_map(
 items=serial_numbers,
 func=fetch_with_cache,
 batch_size=100,
 max_concurrent=10
)

 print(f"Processed {len(units)} units")
 print(f"Cache hit rate: {product_cache.stats.hit_rate:.1%}")
 # Expect 95%+ hit rate on subsequent runs

import asyncio
asyncio.run(process_production_data())

```

## Internal API Usage

### Understanding Internal vs Public APIs

pyWATS provides access to both **public** and **internal** WATS API endpoints:

## Public APIs (Stable):

- Documented and stable endpoints (e.g., `/api/Product`, `/api/Report`)
- Accessed through standard modules: `api.product`, `api.report`, `api.asset`, etc.
- **Guaranteed stability** - Will not change without notice
- **Recommended** for production code

## Internal APIs (Unstable):

- Undocumented endpoints used by WATS frontend (e.g., `/api/internal/UnitFlow`)
- Accessed through the same domain accessor: `api.product`, `api.analytics`, etc.
- Methods are marked with  INTERNAL API in docstrings
-  **MAY CHANGE WITHOUT NOTICE** - Subject to breaking changes
- **Use with caution** - Only when public APIs don't provide needed functionality

## When to Use Internal APIs

Internal APIs fill gaps where public endpoints don't yet exist:

Feature	Method	Why Internal?
Unit Flow Analysis	<code>api.analytics.get_unit_flow()</code>	No public Unit Flow endpoints yet
Box Build Templates	<code>api.product.get_box_build_template()</code>	No public box build management
Asset File Operations	<code>api.asset.upload_blob()</code>	No public file upload/download
Unit Phases (MES)	<code>api.production.get_all_unit_phases()</code>	MES integration not in public API
Process Details	<code>api.process.get_all_processes()</code>	Full process info not in public API

## Using Internal APIs Safely

```
from pywats import pyWATS

api = pyWATS(base_url="...", token="...")

Public API - Use this when available
products = api.product.get_products()

 Internal API - Use only when necessary
This uses internal endpoints that may change (see docstring for warning)
box_build = api.product.get_box_build_template("WIDGET-001", "A")
```

## Best Practices:

- 1. Prefer Public APIs:** Always use public APIs when available
- 2. Isolate Internal Calls:** Wrap internal API calls in your own functions

- 3. Add Error Handling:** Internal APIs may fail differently than public ones
- 4. Document Usage:** Note which parts of your code use internal APIs
- 5. Monitor for Changes:** Subscribe to pyWATS updates for breaking changes

### Example - Isolated Internal API Usage:

```
def get_unit_flow_safely(api, part_number, date_from, date_to):
 """
 Get unit flow data using internal API.

 ! INTERNAL API: This function uses internal WATS endpoints
 that may change without notice.
 """

 try:
 from pywats import UnitFlowFilter

 filter_data = UnitFlowFilter(
 part_number=part_number,
 date_from=date_from,
 date_to=date_to
)

 # Internal API call - wrapped for safety
 result = api.analytics.get_unit_flow(filter_data)
 return result

 except AttributeError:
 # API may have changed
 raise RuntimeError(
 "Unit Flow API has changed. "
 "Please update pyWATS to the latest version."
)

 # Use the wrapped function
try:
 flow = get_unit_flow_safely(api, "WIDGET-001", date_from, date_to)
except RuntimeError as e:
 # Handle API change gracefully
 logging.error(f"Unit Flow not available: {e}")
 # Fall back to alternative approach
```

## Deprecation Warnings

Some internal API methods will emit deprecation warnings:

```
This method is deprecated and will show a warning
bom_xml = api.product.repository.get_bom("WIDGET-001", "A")
DeprecationWarning: ProductRepository.get_bom() uses an internal API endpoint...

Use the unified API method instead
bom_items = api.product.get_bom_items("WIDGET-001", "A")
```

To suppress these warnings during testing (not recommended for production):

```
import warnings

with warnings.catch_warnings():
 warnings.simplefilter("ignore", DeprecationWarning)
 bom = api.product.repository.get_bom("WIDGET-001", "A")
```

## Future-Proofing Your Code

As public APIs become available, migrate away from internal APIs:

```
Current unified API - Internal methods are marked in docstrings
def get_box_build_template(api, part_number, revision):
 # ⚠️ INTERNAL API (see docstring) - may change
 return api.product.get_box_build_template(part_number, revision)

When endpoint becomes public, docstring warning will be removed
Your code continues to work with no changes required
```

### Migration Checklist:

- Monitor pyWATS release notes for new public endpoints
- Test your code with each new pyWATS version
- Keep internal API usage isolated and documented
- Have fallback strategies for critical workflows

## Batch Operations & Pagination

### Batch Operations

Execute multiple API calls concurrently for better performance:

```

from pywats.core import batch_execute, collect_successes, collect_failures

Fetch multiple products in parallel
part_numbers = ["PN-001", "PN-002", "PN-003", "PN-004", "PN-005"]

results = batch_execute(
 keys=part_numbers,
 operation=lambda pn: api.product.get_product(pn),
 max_workers=5 # Concurrent threads (default: 10)
)

Extract successful results
products = collect_successes(results)
print(f"Fetched {len(products)} products successfully")

Check for failures
failures = collect_failures(results)
for key, error in failures:
 print(f"Failed to fetch {key}: {error}")

```

Domain-specific batch methods are also available:

```

Product domain batch methods
product_results = api.product.get_products_batch(["PN-001", "PN-002", "PN-003"])

Fetch multiple revisions
revision_pairs = [("PN-001", "A"), ("PN-001", "B"), ("PN-002", "A")]
revision_results = api.product.get_revisions_batch(revision_pairs)

```

## Pagination

Iterate over large datasets efficiently without loading everything into memory:

```

SCIM: Iterate over all users
for user in api.scim.iter_users(page_size=100):
 print(f"{user.user_name}: {user.display_name}")

With a limit
for user in api.scim.iter_users(page_size=50, max_users=200):
 process_user(user)

With progress tracking
def on_page(page_num, items_so_far, total):
 print(f"Page {page_num}: {items_so_far}/{total} users")

for user in api.scim.iter_users(on_page=on_page):
 sync_to_external_system(user)

```

For custom pagination needs, use the core utilities directly:

```
from pywats.core import paginate, Paginator

Custom pagination with any API
def fetch_page(start_index, count):
 return api.some_api.get_items(start=start_index, count=count)

for item in paginate(
 fetch_page=fetch_page,
 get_items=lambda r: r.items,
 get_total=lambda r: r.total_count,
 page_size=50
):
 process(item)
```

## See Also

---

- Domain Guides - API documentation for each domain
- GUI Configuration - GUI client setup
- Headless Operation - Headless client setup
- Examples - Working code examples

---

Source: [docs/quick-reference.md](#)

# Quick Reference: pyWATS API & Client

---

## API Features

---

### Core Library

- **Async-First Architecture** - Built on `httpx` with native async support
- **Sync Compatibility** - Full sync API via thin wrappers
- **9 Domain Services** - Product, Asset, Report, Production, Analytics, Software, RootCause, Process, SCIM
- **170+ API Endpoints** - Centralized route management
- **Structured Logging** - Configurable verbosity with debug support

## Performance Optimizations

- **Enhanced TTL Caching** - 100x faster cache hits, automatic expiration with background cleanup
- **Connection Pooling** - HTTP/2 multiplexing, 3-5x faster bulk operations
- **Request Batching** - 5-10x faster processing, 95% reduction in server calls
- **MessagePack Serialization** - 50% smaller payloads, 3x faster (optional)

## Offline Queue

- **File-Based Queue** - WSJF format as standard, WSXF/WSTF/ATML conversion
- **Automatic Retry** - Configurable max attempts with exponential backoff
- **Statistics Tracking** - Metadata tracking for queue operations

# Client Features

---

## New Components Created

Component	File	Purpose
<b>pyWATSApplication</b>	src/pywats_client/app.py	Base application without GUI
<b>SettingsManager</b>	src/pywats_client/services/settings_manager.py	Persistent settings storage
<b>SerialNumberManager</b>	src/pywats_client/services/serial_manager.py	Offline serial management
<b>FileMonitor</b>	src/pywats_client/services/file_monitor.py	Folder watching & monitoring
<b>ServiceApplication</b>	src/pywats_client/examples/service_application.py	Complete integration example

## Lines of Code Added

- **Application Layer:** 720 lines
- **Settings Manager:** 480 lines
- **Serial Manager:** 360 lines
- **File Monitor:** 340 lines
- **Example:** 400 lines

- **Documentation:** 1000+ lines
- **Total:** ~2,850 lines of new code

## How to Use

---

### 1. Simple Service (No GUI)

```
from pywats_client import pyWATSApplication, ClientConfig

config = ClientConfig.load("config.json")
app = pyWATSApplication(config)
app.run() # Runs until interrupted
```

### 2. Full Service with All Features

```
from pywats_client.examples.service_application import ServiceApplication

service = ServiceApplication(config_dir=Path("./config"))
service.run()
```

### 3. Async Usage

```
import asyncio
from pywats_client import pyWATSApplication

async def main():
 config = ClientConfig.load("config.json")
 app = pyWATSApplication(config)
 await app.start()
 # Use app...
 await app.stop()

asyncio.run(main())
```

## Key Features

---

### Base Application ( `pyWATSApplication` )

- Lifecycle management (start, stop, restart)
- Service orchestration
- Status tracking & callbacks

- Error handling & callbacks
- Multi-instance support

## **Settings Management**

- JSON file storage
- Validation
- Auto-backup
- Monitor folder config
- Converter config

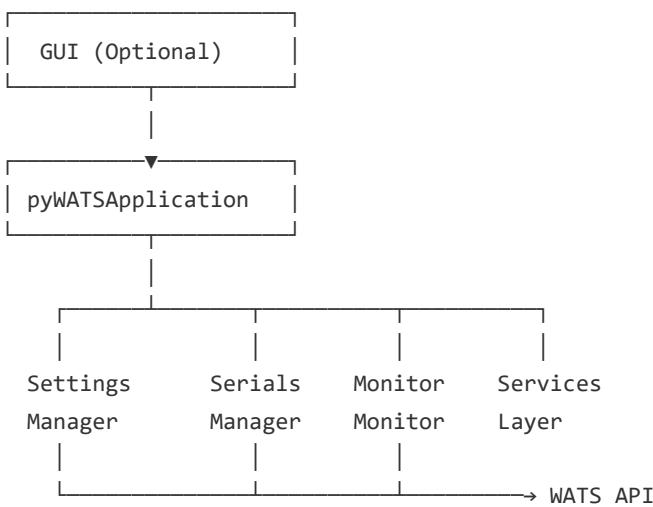
## **Serial Management**

- Reserve serials offline
- Persistent storage
- Pool statistics
- Usage tracking
- Auto-replenish indicators

## **File Monitoring**

- Watch folders for files
- Multiple rules
- Auto-conversion
- Debouncing
- Callback events

# Architecture



## Configuration (settings.json)

```
{
 "server_url": "https://python.wats.com",
 "api_token": "your-token",
 "auto_upload_reports": true,
 "auto_upload_interval": 60,
 "monitor_folders": [
 {
 "path": "./uploads",
 "converter_type": "csv",
 "auto_upload": true,
 "delete_after_convert": true
 }
]
}
```

## Git Branch

Branch: pyWATS\_Service-and-Client  
Commits: 3 new commits  
Status: Ready for review

## Latest Commits

1.  **4041c20** - Phase 1 status and next steps documentation
2.  **0ab8ea8** - Complete service application example

3.  **14ec311** - Core refactoring with all new components

## What's Next

---

### Phase 2: Error Handling

- Integrate WATS API ErrorHandler/ErrorMode
- Better error tracking and recovery

### Phase 3: GUI Update

- Refactor GUI to use pyWATSApplication
- Remove business logic duplication

### Phase 4: System Integration

- Windows Service wrapper
- systemd service configuration
- Docker setup

### Phase 5: Testing

- Unit tests for new services
- Integration tests
- End-to-end scenarios

### Phase 6: Packaging

- Installable packages
- Cross-platform builds
- Release distribution

# File Structure

---

New/Modified Files:

```
|── src/pywats_client/
| ├── app.py [NEW]
| ├── __init__.py [UPDATED]
| ├── services/
| | ├── settings_manager.py [NEW]
| | ├── serial_manager.py [NEW]
| | └── file_monitor.py [NEW]
| └── examples/
| └── service_application.py [NEW]
├── ARCHITECTURE_REFACTORING.md [NEW]
└── CLIENT_SERVICE_REFACTORING_STATUS.md [NEW]
```

## Documentation

---

- **Detailed Architecture:** See `ARCHITECTURE_REFACTORING.md`
- **Phase Status:** See `CLIENT_SERVICE_REFACTORING_STATUS.md`
- **API Docs:** See docstrings in source files
- **Examples:** See `src/pywats_client/examples/service_application.py`

## Key Design Principles

---

1. **Separation of Concerns** - Base app has no GUI dependencies
2. **Async-First** - All I/O is async
3. **Callback-Based Events** - Decoupled components
4. **Persistent State** - Settings and serials stored locally
5. **Error Resilient** - Auto-reconnection and graceful degradation

## Test Status

---

- All 90 existing tests still pass
- Zero errors in new code
- Backward compatible

# Deployment Options

---

## 1. Service/Daemon

```
python -m pywats_client.gui.app --headless
```

## 2. Windows Service

```
Wrapper coming in Phase 4
```

## 3. systemd Service

```
Configuration coming in Phase 4
```

## 4. Docker Container

```
Dockerfile coming in Phase 4
```

## 5. Custom Application

```
from pywats_client import pyWATSApplication
... integrate as needed
```

# Troubleshooting

---

## Application won't start

- Check `settings.json` is valid
- Check `server_url` is accessible
- Check instance not already running

## Files not being converted

- Check `monitor_folders` configured
- Check folder path exists
- Check converter\_type is correct

- Look at logs for errors

## Serials depleted

- Check `auto_reserve_serials` is enabled
- Ensure online to replenish
- Check `reserve_count` setting

## Support Resources

---

Topic	Location
Architecture	ARCHITECTURE_REFACTORING.md
Status	CLIENT_SERVICE_REFACTORING_STATUS.md
Examples	src/pywats_client/examples/service_application.py
API Reference	Docstrings in source files
Tests	tests/test_*.py files

## Summary

---

### Phase 1 Complete

- Base application layer fully implemented
- Settings persistence working
- Serial number management ready
- File monitoring operational
- Complete example application provided
- Comprehensive documentation created
- All tests passing
- Ready for review and Phase 2

---

**Branch:** pyWATS\_Service-and-Client

**Status:** Complete and Ready

**Next:** Error Handling Integration (Phase 2)

# Installation

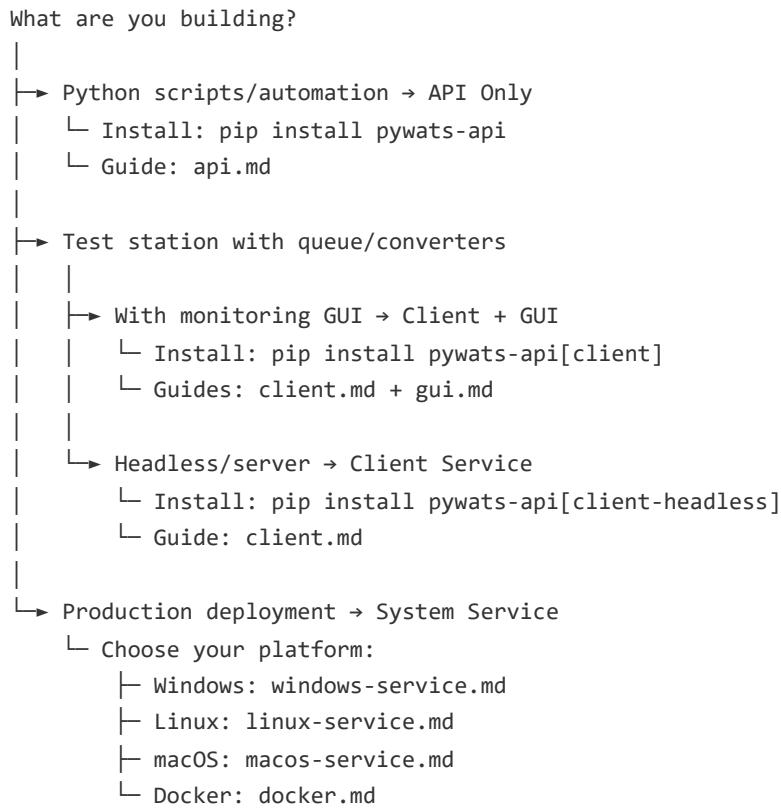
Source: [docs/installation/README.md](#)

## Installation & Deployment Documentation

This directory contains guides for installing PyWATS components and deploying the client service.

### ⌚ What Do You Need?

Use this decision tree to find the right installation guide:



# Installation by Component

## API Library

**API Installation** - Python SDK for direct WATS integration

- pip package only (~5 MB)
- No background services
- Use from scripts and applications

```
pip install pywats-api
```

## Client Service

**Client Service** - Background service with queue and converters

- Report queue with retry
- File watching and converters
- Offline support

```
With GUI
pip install pywats-api[client]

Headless (no GUI)
pip install pywats-api[client-headless]
```

## GUI Application

**GUI Application** - Desktop app for monitoring and configuration

- Real-time queue monitoring
- Configuration interface
- Log viewer

```
pip install pywats-api[client]
```

# Service Deployment

Install the client as a system service for automatic startup:

Platform	Guide	Method
Windows	windows-service.md	NSSM / sc.exe
Linux	linux-service.md	systemd
macOS	macos-service.md	launchd
Docker	docker.md	Container

## Quick Comparison

Feature	API Only	Client Headless	Client + GUI
Size	~5 MB	~8 MB	~150 MB
Python SDK	✓	✓	✓
Report Queue	-	✓	✓
Converters	-	✓	✓
File Watching	-	✓	✓
GUI	-	-	✓
Use Case	Scripts	Servers, Pi	Desktop

## Quick Start

### Developers/Integrators

```
pip install pywats-api
```

```
from pywats import pyWATS
api = pyWATS(base_url="...", token="...")
```

## Test Stations (Desktop)

```
pip install pywats-api[client]
python -m pywats_client service
python -m pywats_client gui # In another terminal
```

## Test Stations (Headless)

```
pip install pywats-api[client-headless]
pywats-client config init
pywats-client start
```

## See Also

- [..../INDEX.md](#) - Main documentation index
- [..../getting-started.md](#) - Complete tutorial
- [..../client-architecture.md](#) - Client service internals
- [..../env-variables.md](#) - Environment variable reference

---

*Source: docs/installation/api.md*

# PyWATS API Installation

Install the core PyWATS API library for direct integration with WATS from your Python applications.

## Overview

The API package provides:

- Python SDK for all WATS domains (Report, Product, Production, Asset, etc.)
- Sync and async client support
- Data validation with Pydantic 2.0+
- No GUI dependencies

**Best for:** Scripts, automation, server-side integrations, custom applications.

# Installation

```
pip install pywats-api
```

## Requirements:

- Python 3.10+
- ~5 MB disk space

## Dependencies (automatically installed):

- `httpx` - HTTP client
- `pydantic` - Data validation
- `python-dateutil` - Date utilities

# Quick Start

## Basic Connection

```
from pywats import pyWATS

Initialize client
api = pyWATS(
 base_url="https://your-server.wats.com",
 token="your_base64_encoded_token"
)

Test connection
if api.test_connection():
 print("Connected to WATS!")
```

## Using Environment Variables

Set credentials once, use everywhere:

```
Windows
set PYWATS_SERVER_URL=https://your-server.wats.com
set PYWATS_API_TOKEN=your_base64_encoded_token

Linux/macOS
export PYWATS_SERVER_URL=https://your-server.wats.com
export PYWATS_API_TOKEN=your_base64_encoded_token
```

Then in Python:

```
from pywats import pyWATS

Reads from environment automatically
api = pyWATS()
```

## Async Client

For high-performance applications:

```
import asyncio
from pywats import pyWATS

async def main():
 async with pyWATS(async_mode=True) as api:
 products = await api.product.get_products()
 print(f"Found {len(products)} products")

asyncio.run(main())
```

## API Domains

The API is organized into domain modules:

Domain	Import	Use Case
<b>Report</b>	api.report	Create/query test reports (UUT/UUR)
<b>Product</b>	api.product	Manage products, revisions, BOMs
<b>Production</b>	api.production	Serial numbers, unit lifecycle
<b>Asset</b>	api.asset	Equipment tracking, calibration
<b>Analytics</b>	api.analytics	Yield analysis, statistics
<b>Software</b>	api.software	Package distribution
<b>RootCause</b>	api.rootcause	Issue tracking, defects
<b>Process</b>	api.process	Operation types, processes
<b>SCIM</b>	api.scim	User provisioning

## Example: Create a Test Report

```
from pywats import pyWATS
from pywats.report import UUTReport

api = pyWATS()

Create report
report = UUTReport(
 part_number="PCB-001",
 serial_number="SN12345",
 operation_code="ICT",
 result="P"
)

Add a numeric measurement
report.add_numeric_limit_step(
 name="Voltage Check",
 value=3.3,
 low_limit=3.0,
 high_limit=3.6,
 unit="V"
)

Submit
result = api.report.submit_uut_report(report)
print(f"Report ID: {result.id}")
```

## Authentication

### Token Generation

1. Log into WATS web interface
2. Navigate to **Settings → API Access**
3. Generate a new API token
4. Copy the base64-encoded token

### Token Format

The token is a base64-encoded string containing your credentials:

```
import base64

Create token from username:password
credentials = "username:password"
token = base64.b64encode(credentials.encode()).decode()
print(token) # dXNlcjIyMzQwMjUxMjIwMjAx
```

## Configuration Options

### Client Initialization

```
from pywats import pyWATS

api = pyWATS(
 base_url="https://your-server.wats.com", # WATS server URL
 token="...", # Auth token
 timeout=30, # Request timeout (seconds)
 verify_ssl=True, # SSL certificate verification
 async_mode=False, # Use async client
)
```

### Environment Variables

Variable	Description	Default
PYWATS_SERVER_URL	WATS server base URL	Required
PYWATS_API_TOKEN	Base64-encoded auth token	Required
PYWATS_TIMEOUT	Request timeout (seconds)	30
PYWATS_VERIFY_SSL	Verify SSL certificates	true
PYWATS_LOG_LEVEL	Logging level	INFO

## Upgrading

```
pip install --upgrade pywats-api
```

Check current version:

```
pip show pywats-api
```

Or in Python:

```
import pywats
print(pywats.__version__)
```

## Troubleshooting

### Import Errors

```
Verify installation
pip show pywats-api

Check Python version
python --version # Should be 3.10+

Reinstall
pip uninstall pywats-api
pip install pywats-api
```

### Connection Issues

```
from pywats import pyWATS

api = pyWATS(
 base_url="https://your-server.wats.com",
 token="your_token"
)

Enable debug logging
import logging
logging.basicConfig(level=logging.DEBUG)

Test connection
try:
 api.test_connection()
except Exception as e:
 print(f"Connection failed: {e}")
```

## SSL Certificate Errors

For development/testing only:

```
api = pyWATS(
 base_url="https://... ",
 token="... ",
 verify_ssl=False # ⚠️ Not for production!
)
```

## Next Steps

- **Getting Started Guide** - Comprehensive tutorial
- **Quick Reference** - Common patterns and snippets
- **Domain Documentation** - Detailed API reference

## Need More?

If you need...	Install...	Guide
Background service with queue	<code>pip install pywats-api[client-headless]</code>	Service Guide
Desktop GUI for monitoring	<code>pip install pywats-api[client]</code>	GUI Guide
Development tools	<code>pip install pywats-api[dev]</code>	Getting Started

## See Also

- **../INDEX.md** - Main documentation index
- **../architecture.md** - System architecture overview
- **../env-variables.md** - Environment variable reference

# PyWATS Client Service

The PyWATS Client Service is a background process that handles automated test report processing, queuing, and upload to WATS.

## Overview

The client service provides:

- **Report Queue** - Reliable queuing with retry on failure
- **Offline Support** - Queue reports when disconnected, upload when online
- **Converters** - Transform test equipment output to WATS format
- **File Watching** - Auto-detect new reports in watch folders
- **Multi-Instance** - Run separate instances for different stations

**Best for:** Test station automation, production environments, embedded systems.

## Installation

### With GUI (Desktop Stations)

```
pip install pywats-api[client]
```

Includes desktop GUI for configuration and monitoring. See GUI Guide.

### Headless (Servers, Embedded)

```
pip install pywats-api[client-headless]
```

No GUI dependencies - CLI and HTTP API only. Ideal for:

- Linux servers
- Raspberry Pi
- Embedded systems
- Docker containers

## Requirements:

- Python 3.10+
  - ~8 MB disk space (headless) / ~150 MB (with GUI)
- 

# Quick Start

---

## Start the Service

```
Default instance
python -m pywats_client service

Named instance
python -m pywats_client service --instance-id station1
```

## First-Time Configuration

### Interactive setup:

```
pywats-client config init
```

### Non-interactive:

```
pywats-client config init \
 --server-url https://wats.yourcompany.com \
 --username your-username \
 --password your-password \
 --station-name ICT-STATION-01 \
 --non-interactive
```

## Verify Connection

```
pywats-client status
```

---

## Architecture

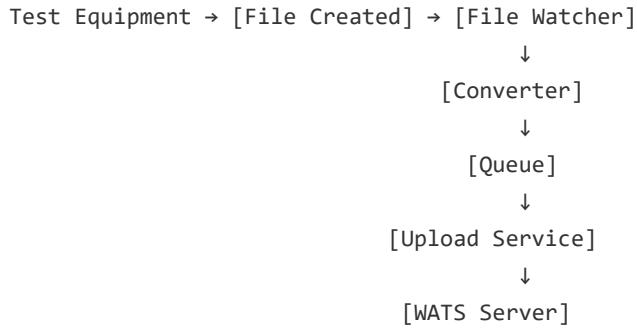
---

## Background Services

The client runs several background services:

Service	Purpose
<b>Connection Monitor</b>	Watches WATS server connectivity
<b>Queue Manager</b>	Manages report queue lifecycle
<b>File Watcher</b>	Detects new files in watch folders
<b>Upload Service</b>	Sends reports to WATS
<b>Software Service</b>	Downloads software packages (optional)

## Report Processing Flow



## File Organization

### Data Directories

#### Windows (Production):

```

C:\ProgramData\Virinco\pyWATS\
├── config.json # Configuration
├── logs\ # Service logs
└── queue\ # Report queue
 ├── pending\ # Waiting for upload
 ├── processing\ # Currently uploading
 ├── completed\ # Successfully uploaded
 └── failed\ # Failed uploads
└── converters\ # Custom converters
└── data\ # Software packages

```

#### Windows (User Development):

```
%APPDATA%\pyWATS_Client\
```

## Linux/macOS:

```
~/.config/pywats_client/ # User
/var/lib/pywats/ # System service
```

## Queue Folder Contents

Folder	Purpose	Auto-Cleanup
pending/	Reports waiting for upload	No
processing/	Currently uploading	No
completed/	Successfully uploaded	After 7 days
failed/	Failed uploads	After 30 days

## Configuration

### Configuration File

config.json :

```
{
 "server_url": "https://wats.yourcompany.com",
 "api_token": "...",
 "station_name": "ICT-STATION-01",
 "station_location": "Production Line A",

 "queue": {
 "watch_folders": ["C:\\TestReports"],
 "upload_interval": 10,
 "retry_attempts": 3,
 "retry_delay": 60
 },
 "converters": {
 "enabled": ["WATSStandardXMLConverter", "TeradyneICTConverter"],
 "auto_detect": true
 },
 "logging": {
 "level": "INFO",
 "max_size_mb": 10,
 "backup_count": 5
 }
}
```

## CLI Configuration

```
View all settings
pywats-client config show

Get specific value
pywats-client config get queue.upload_interval

Set value
pywats-client config set queue.upload_interval 30

Add watch folder
pywats-client config add-watch-folder "C:\\TestReports\\Station1"
```

## Environment Variables

Override config via environment:

Variable	Description
PYWATS_SERVER_URL	WATS server URL
PYWATS_API_TOKEN	Auth token
PYWATS_STATION_NAME	Station identifier
PYWATS_LOG_LEVEL	Logging level
PYWATS_WATCH_FOLDERS	Colon-separated paths

## Converters

---

Converters transform test equipment output into WATS format.

### Built-in Converters

#### WATS Standard Formats:

Converter	Format	File Patterns
WATSStandardXMLConverter	WSXF/WRML	*.xml
WATSStandardJsonConverter	WSJF	*.json
WATSStandardTextConverter	WSTF	*.txt

#### Industry Standards:

Converter	Standard	File Patterns	Notes
ATMLConverter	IEEE ATML (1671/1636.1)	*.xml , *.atml	ATML 2.02, 5.00, 6.01 + TestStand AddOn

#### Test Equipment:

Converter	Equipment	File Patterns
TeradyneICTConverter	Teradyne i3070	*.txt , *.log
TeradyneSpectrumICTConverter	Teradyne Spectrum	*.txt , *.log
SeicaXMLConverter	Seica Flying Probe	*.xml
KlippelConverter	Klippel Audio/Acoustic	*.txt + data folder
SPEAConverter	SPEA ATE	*.txt
XJTAGConverter	XJTAG Boundary Scan	*.zip

## Special:

Converter	Purpose
AIConverter	Auto-detects file type and delegates to best matching converter

## Custom Converters

Place custom converters in the converters folder:

```
converters/my_converter.py
from pywats_client.converters import BaseConverter, ConverterInfo
from pywats.report import UUTReport

class MyConverter(BaseConverter):
 @classmethod
 def get_info(cls) -> ConverterInfo:
 return ConverterInfo(
 name="MyConverter",
 description="Converts my test equipment output",
 file_patterns=["*.myext"],
 version="1.0.0"
)

 def convert(self, file_path: str) -> UUTReport:
 # Parse file and create report
 report = UUTReport(
 part_number="...",
 serial_number="...",
 operation_code="TEST",
 result="P"
)
 return report
```

See LLM Converter Guide for detailed examples.

## Queue Management

### CLI Commands

```
View queue status
pywats-client queue status

List pending reports
pywats-client queue list

Retry failed reports
pywats-client queue retry-failed

Clear completed reports
pywats-client queue clear-completed

Manual upload
pywats-client upload --file /path/to/report.xml
```

### Queue API

For programmatic access:

```
from pywats_client.core.queue import ReportQueue

queue = ReportQueue(config_path)

Add report
queue.add("path/to/report.xml")

Get status
status = queue.get_status()
print(f"Pending: {status.pending_count}")

Process queue
await queue.process_all()
```

## Multi-Instance Support

Run separate instances for different stations:

```
Instance 1
pywats-client --instance station1 config init
pywats-client --instance station1 service

Instance 2
pywats-client --instance station2 config init
pywats-client --instance station2 service
```

Each instance has separate:

- Configuration file
  - Queue folder
  - Log file
  - Watch folders
- 

## Headless Mode

---

### CLI Control

```
Start foreground
pywats-client start

Start as daemon (Linux/macOS)
pywats-client start --daemon

Check status
pywats-client status

Stop daemon
pywats-client stop

Restart
pywats-client restart
```

### HTTP API Control

```
Start with HTTP API
pywats-client start --api --api-port 8765
```

Available endpoints:

Endpoint	Method	Description
/status	GET	Service status
/queue	GET	Queue status
/queue/pending	GET	List pending reports
/restart	POST	Restart service
/stop	POST	Stop service
/config	GET	View configuration

```
Examples
curl http://localhost:8765/status
curl http://localhost:8765/queue
curl -X POST http://localhost:8765/restart
```

## Running as System Service

For production deployments, run as a system service:

- **Windows Service** - NSSM setup, auto-start
- **Linux Service** - Systemd configuration
- **macOS Service** - Launchd daemon
- **Docker** - Container deployment

# Troubleshooting

## Service Won't Start

```
Check Python version
python --version # Should be 3.10+

Check installation
pip show pywats-api

View logs
cat ~/.config/pywats_client/pywats_client.log
```

## Reports Not Uploading

### 1. Check connection:

```
bash pywats-client status
```

### 2. Check queue:

```
bash pywats-client queue status
```

### 3. Enable debug logging:

```
bash pywats-client config set logging.level DEBUG pywats-client restart
```

### 4. Manual upload test:

```
bash pywats-client upload --file /path/to/report.xml --verbose
```

## Converter Not Detecting Files

### 1. Check watch folders:

```
bash pywats-client config get queue.watch_folders
```

### 2. Check file patterns:

```
bash pywats-client converters list
```

### 3. Test converter manually:

```
bash pywats-client convert --file /path/to/file.txt --converter MyConverter
```

## Finding Configuration

```
Show config path
pywats-client config show --format json | grep config_path

Or check default locations:
Windows: %APPDATA%\pyWATS_Client\config.json
Linux/macOS: ~/.config/pywats_client/config.json
```

# Security

---

## Credential Storage

- Passwords encrypted using platform-specific encryption
- **Windows:** DPAPI (Data Protection API)
- **Linux/macOS:** System keyring or file encryption

## Network Security

- All WATS communication uses HTTPS
- Credentials never logged
- API tokens rotated on password change

## File Permissions

```
Linux/macOS - restrict config access
chmod 600 ~/.config/pywats_client/config.json
```

## See Also

---

- **API Installation** - SDK-only installation
- **GUI Guide** - Desktop application
- **../client-architecture.md** - Architecture details
- **../ilm-converter-guide.md** - Writing converters
- **../getting-started.md** - Complete tutorial

---

Source: *docs/installation/gui.md*

# PyWATS GUI Application

---

The PyWATS GUI provides a desktop application for monitoring and configuring your WATS client service.

# Overview

---

The GUI application offers:

- **Real-time Monitoring** - View queue status, upload progress, connection state
- **Configuration Interface** - Configure server, converters, and settings
- **Log Viewer** - Monitor application and service logs
- **Converter Management** - Enable/disable and configure converters

**Important:** The GUI is a companion application for the client service. The service handles the actual work (queue processing, uploads); the GUI provides visibility and configuration.

---

## Installation

---

```
pip install pywats-api[client]
```

### Requirements:

- Python 3.10+
- Display/monitor (X11/Wayland on Linux)
- ~150 MB disk space

### Dependencies (automatically installed):

- PySide6 - Qt6 GUI framework
- watchdog - File monitoring
- aiofiles - Async file operations
- Plus all API dependencies

---

## Quick Start

---

### Starting the Service and GUI

The recommended workflow is to run the service first, then connect the GUI:

#### Step 1: Start the service

```
python -m pywats_client service --instance-id default
```

#### Step 2: Launch GUI (in another terminal)

```
python -m pywats_client gui --instance-id default
```

The GUI will connect to the running service via IPC.

## First-Time Setup

1. Launch the GUI
2. Go to **Setup** tab
3. Enter your WATS server details:
4. **Server URL:** <https://your-server.wats.com>
5. **Username:** Your WATS username
6. **Password:** Your WATS password
7. **Station Name:** Identifier for this test station
8. Click **Test Connection**
9. Click **Save**

---

## GUI Tabs

---

### Dashboard

Main overview showing:

- Connection status (connected/disconnected)
- Queue statistics (pending, processing, completed, failed)
- Recent uploads with timestamps
- Service health indicators

### Setup

Configure WATS server connection:

- Server URL
- Credentials
- Station name and location
- Connection test button

### Queue

View and manage the report queue:

- Pending reports waiting for upload
- Processing status

- Failed reports with error details
- Retry/delete options

## Converters

Manage report converters:

- View installed converters
- Enable/disable converters
- Configure converter settings
- View converter status and errors

## Logs

Real-time log viewer:

- Filter by log level (DEBUG, INFO, WARNING, ERROR)
- Search functionality
- Auto-scroll toggle
- Export logs

## Software

Software distribution panel (if enabled):

- Available packages
- Download status
- Version information

---

# Configuration

---

## GUI Settings

The GUI stores its own settings separately from the service:

**Windows:**

```
%APPDATA%\pyWATS_Client\gui_settings.json
```

**Linux/macOS:**

```
~/.config/pywats_client/gui_settings.json
```

## Customizable Options

```
{
 "window_geometry": {
 "width": 1200,
 "height": 800,
 "x": 100,
 "y": 100
 },
 "theme": "system",
 "log_viewer": {
 "max_lines": 10000,
 "auto_scroll": true,
 "show_timestamps": true
 },
 "refresh_interval": 1000,
 "notifications": {
 "upload_complete": true,
 "upload_failed": true,
 "connection_lost": true
 }
}
```

## Themes

The GUI supports system theme detection:

```
Force light theme
python -m pywats_client gui --theme light

Force dark theme
python -m pywats_client gui --theme dark

Use system preference (default)
python -m pywats_client gui --theme system
```

## Command Line Options

```
python -m pywats_client gui [OPTIONS]
```

Options:

```
--instance-id TEXT Client instance to connect to (default: "default")
--config-path PATH Path to config file
--theme TEXT Theme: light, dark, system
--minimized Start minimized to system tray
--help Show help message
```

## Examples

```
Connect to default instance
python -m pywats_client gui

Connect to specific instance
python -m pywats_client gui --instance-id station2

Start minimized
python -m pywats_client gui --minimized

Custom config location
python -m pywats_client gui --config-path /path/to/config.json
```

## System Tray

The GUI can minimize to the system tray:

- **Double-click** tray icon to restore window
- **Right-click** for context menu:
  - Show/Hide window
  - View status
  - Open logs folder
  - Exit

## Tray Notifications

When minimized, the GUI shows notifications for:

- Upload completed
- Upload failed
- Connection lost/restored
- Service stopped

Notifications can be disabled in settings.

## Troubleshooting

### GUI Won't Start

**Check Qt installation:**

```
python -c "from PySide6 import QtWidgets; print('Qt OK')"
```

## If import fails:

```
pip uninstall PySide6
pip install PySide6
```

## Linux: Check display server:

```
echo $DISPLAY # Should show :0 or similar
```

## GUI Can't Connect to Service

### 1. Verify service is running:

```
bash python -m pywats_client status --instance-id default
```

### 2. Start service if needed:

```
bash python -m pywats_client service --instance-id default
```

### 3. Check instance ID matches:

```
bash # Both must use same instance-id
python -m pywats_client service --instance-id mystation
python -m pywats_client gui --instance-id mystation
```

## Blank or Frozen UI

### 1. Check log file for errors:

2. Windows: %APPDATA%\pyWATS\_Client\pywats\_client.log

3. Linux: ~/.config/pywats\_client/pywats\_client.log

### 4. Try resetting GUI settings:

```
bash # Remove GUI settings (will use defaults) rm ~/.config/pywats_client/gui_settings.json
```

## High DPI Display Issues

```
Force DPI scaling
export QT_AUTO_SCREEN_SCALE_FACTOR=1
python -m pywats_client gui

Or set specific scale
export QT_SCALE_FACTOR=1.5
python -m pywats_client gui
```

## Keyboard Shortcuts

---

Shortcut	Action
Ctrl+Q	Quit application
Ctrl+L	Focus log search
Ctrl+R	Refresh all panels
Ctrl+,	Open settings
F5	Refresh queue
F11	Toggle fullscreen
Ctrl+M	Minimize to tray

---

## Without GUI (Headless Alternative)

---

If you don't need a GUI, install the headless version instead:

```
pip install pywats-api[client-headless]
```

This provides all service functionality via CLI and HTTP API:

```
Check status
pywats-client status

View queue
pywats-client queue list

Control via HTTP API
curl http://localhost:8765/status
```

See Client Service Guide for headless operation.

---

## See Also

---

- **Client Service Guide** - Background service documentation

- **Windows Service** - Auto-start on Windows
  - **Linux Service** - Systemd service setup
  - **../getting-started.md** - Complete tutorial
  - **../client-architecture.md** - Architecture details
- 

*Source: docs/installation/docker.md*

# Docker Deployment Guide for pyWATS

---

This guide covers deploying pyWATS using Docker for various use cases.

## Table of Contents

---

- Quick Start
  - Available Images
  - Production Deployment
  - Development Setup
  - Configuration
  - Monitoring & Troubleshooting
  - Advanced Usage
- 

## Quick Start

---

### 1. Prerequisites

- Docker 20.10+ and Docker Compose 2.0+
- WATS server credentials

## 2. Initial Setup

```
Clone the repository
git clone https://github.com/olreppe/pyWATS.git
cd pyWATS

Create environment file
cp .env.example .env

Edit .env and add your WATS credentials
nano .env
```

## 3. Create Required Directories

```
mkdir -p watch output archive config
```

## 4. Create Client Configuration

Create config/client\_config.json :

```
{
 "wats": {
 "base_url": "https://wats.yourcompany.com",
 "token": "your_base64_token"
 },
 "converters": {
 "enabled": ["csv", "json", "xml"],
 "watch_directory": "/app/watch",
 "output_directory": "/app/output",
 "archive_directory": "/app/archive"
 },
 "logging": {
 "level": "INFO",
 "file": "/app/logs/pywats_client.log"
 }
}
```

## 5. Start the Client

```
Start headless client
docker-compose up -d client

View logs
docker-compose logs -f client

Check status
docker-compose ps
```

# Available Images

The Dockerfile provides multiple build targets:

## 1. API Only ( api )

Minimal image with just the pyWATS API library.

```
Build
docker build --target api -t pywats-api .

Run Python with pyWATS
docker run -it pywats-api python
```

### Use Cases:

- Python scripts that use pyWATS API
- Custom applications
- Lambda/serverless functions

## 2. Headless Client ( client-headless )

Client without GUI for servers and embedded systems.

```
Build
docker build --target client-headless -t pywats-client .

Run with config
docker run -d \
-v $(pwd)/config:/app/config:ro \
-v $(pwd)/watch:/app/watch \
-v $(pwd)/output:/app/output \
-e WATS_BASE_URL=https://wats.example.com \
-e WATS_TOKEN=your_token \
pywats-client
```

### Use Cases:

- Production test data ingestion
- Automated test report uploads
- Headless test stations
- Raspberry Pi deployments

## 3. Development ( dev )

Full development environment with all dependencies.

```
Start dev container
docker-compose --profile dev up -d dev

Attach to container
docker-compose exec dev bash

Run tests
docker-compose exec dev pytest

Build docs
docker-compose exec dev sphinx-build docs/api docs/_build/html
```

## Production Deployment

### Docker Compose (Recommended)

```
1. Configure environment
cp .env.example .env
nano .env

2. Create config
mkdir -p config watch output archive
nano config/client_config.json

3. Start services
docker-compose up -d client

4. Verify
docker-compose logs client
docker-compose ps
```

### Kubernetes

Example deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: pywats-client
spec:
 replicas: 1
 selector:
 matchLabels:
 app: pywats-client
 template:
 metadata:
 labels:
 app: pywats-client
 spec:
 containers:
 - name: client
 image: pywats-client:latest
 env:
 - name: WATS_BASE_URL
 valueFrom:
 secretKeyRef:
 name: pywats-secrets
 key: base-url
 - name: WATS_TOKEN
 valueFrom:
 secretKeyRef:
 name: pywats-secrets
 key: token
 volumeMounts:
 - name: config
 mountPath: /app/config
 readOnly: true
 - name: watch
 mountPath: /app/watch
 - name: output
 mountPath: /app/output
 - name: logs
 mountPath: /app/logs
 resources:
 requests:
 memory: "256Mi"
 cpu: "500m"
 limits:
 memory: "1Gi"
 cpu: "2"
 volumes:
 - name: config
 configMap:
 name: pywats-config
 - name: watch
 persistentVolumeClaim:
 claimName: pywats-watch
 - name: output
 persistentVolumeClaim:
 claimName: pywats-output
```

```
- name: logs
 persistentVolumeClaim:
 claimName: pywats-logs
```

## Docker Swarm

```
Initialize swarm
docker swarm init

Deploy stack
docker stack deploy -c docker-compose.yml pywats

Scale service
docker service scale pywats_client=3

View logs
docker service logs -f pywats_client
```

## Configuration

### Environment Variables

#### Required

- WATS\_BASE\_URL - WATS server URL
- WATS\_TOKEN - Base64-encoded credentials

#### Optional

- PYWATS\_LOG\_LEVEL - Logging level (DEBUG, INFO, WARNING, ERROR)
- PYWATS\_HEADLESS - Run in headless mode (default: 1 in Docker)
- PYWATS\_CONFIG\_DIR - Configuration directory (default: /app/config)
- PYWATS\_DATA\_DIR - Data directory (default: /app/data)
- PYWATS\_LOG\_DIR - Log directory (default: /app/logs)

## Volume Mounts

Mount Point	Purpose	Recommended
/app/config	Configuration files	Read-only in production
/app/watch	Incoming test data	Writable
/app/output	Converted reports	Writable
/app/archive	Processed files	Writable
/app/logs	Application logs	Persistent volume
/app/data	State/queue data	Persistent volume

## Health Checks

All images include health checks:

```
Check container health
docker inspect --format='{{.State.Health.Status}}' pywats-client

View health check logs
docker inspect --format='{{range .State.Health.Log}}{{.Output}}{{end}}' pywats-client
```

## Monitoring & Troubleshooting

### View Logs

```
Real-time logs
docker-compose logs -f client

Last 100 lines
docker-compose logs --tail=100 client

Specific time range
docker-compose logs --since 2024-01-01T00:00:00 client
```

## Check Status

```
Container status
docker-compose ps

Resource usage
docker stats pywats-client

Health status
docker inspect --format='{{.State.Health.Status}}' pywats-client
```

## Common Issues

### Container exits immediately

```
Check logs
docker-compose logs client

Common causes:
1. Missing WATS_BASE_URL or WATS_TOKEN
2. Invalid configuration in config/client_config.json
3. Permission issues on mounted directories
```

### Cannot connect to WATS server

```
Test network connectivity
docker-compose exec client ping wats.yourcompany.com

Test WATS API
docker-compose exec client python -c "
from pywats import pyWATS
api = pyWATS(base_url='https://wats.example.com', token='...')
print(api.test_connection())
"
```

### Permission denied errors

```
Fix directory permissions (host)
chmod -R 777 watch output archive logs

Or run with specific user ID
docker-compose run --user $(id -u):$(id -g) client
```

## Debugging

```
Start interactive shell
docker-compose exec client bash

Or start new container with shell
docker-compose run --rm client bash

Run Python interactively
docker-compose exec client python
```

## Advanced Usage

### Multi-Stage Builds

Build only what you need:

```
API only (smallest)
docker build --target api -t pywats-api:latest .

Headless client
docker build --target client-headless -t pywats-client:latest .

Development (largest)
docker build --target dev -t pywats-dev:latest .
```

### Custom Configuration

Override defaults with `docker-compose.override.yml`:

```
version: '3.8'
services:
 client:
 environment:
 PYWATS_LOG_LEVEL: DEBUG
 volumes:
 - /custom/watch:/app/watch
 - /custom/output:/app/output
```

### Resource Limits

Adjust in `docker-compose.yml`:

```
deploy:
 resources:
 limits:
 cpus: '4' # Max CPUs
 memory: 2G # Max memory
 reservations:
 cpus: '1' # Min CPUs
 memory: 512M # Min memory
```

## Network Configuration

```
Create custom network
docker network create --driver bridge pywats-net

Run with custom network
docker run -d --network pywats-net pywats-client
```

## CI/CD Integration

GitHub Actions example:

```
name: Build Docker Image

on:
 push:
 branches: [main]

jobs:
 build:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v3

 - name: Build image
 run: docker build --target client-headless -t pywats-client:latest .

 - name: Test image
 run: |
 docker run --rm pywats-client python -c "import pywats; print('OK')"

 - name: Push to registry
 run: |
 echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USERNAME }}" --
password-stdin
 docker push pywats-client:latest
```

# Security Considerations

---

1. **Never commit .env files** - Use secrets management
  2. **Use read-only config mounts** - Prevent container from modifying config
  3. **Run as non-root** - All images use non-root user (uid 1000)
  4. **Scan images** - docker scan pywats-client
  5. **Use HTTPS** - Always use TLS for WATS connections
  6. **Rotate credentials** - Update WATS\_TOKEN regularly
  7. **Network isolation** - Use Docker networks to restrict access
- 

## Support

---

- **Documentation:** <https://github.com/olreppe/pyWATS/tree/main/docs>
  - **Issues:** <https://github.com/olreppe/pyWATS/issues>
  - **Email:** support@virinco.com
- 

**Last Updated:** January 23, 2026

**Version:** 0.1.0b34

---

Source: docs/installation/windows-service.md

# Windows Service Installation

---

This guide explains how to install pyWATS Client as a Windows Service that auto-starts on system boot.

## Overview

---

The pyWATS Client can run as a Windows Service in the background, automatically starting when Windows boots. This is the recommended setup for production environments.

### Folder Structure:

- **Installation:** C:\Program Files\Virinco\pyWATS\ (binaries)
- **Data/Config:** C:\ProgramData\Virinco\pyWATS\ (configuration, logs, queues)
- **Service Name:** pyWATS\_Service (appears in Task Manager/Services)

This matches the existing WATS Client installation pattern.

## Prerequisites

---

### Required

- Windows 10/11 or Windows Server 2016+
- Python 3.10 or later
- Administrator privileges

### Recommended: NSSM (Non-Sucking Service Manager)

NSSM provides the best Windows Service experience with:

- Easy service installation/removal
- Automatic log rotation
- Crash recovery
- Better process management

#### Download NSSM:

1. Visit: <https://nssm.cc/download>
2. Download the latest version (2.24+)
3. Extract `nssm.exe` to `C:\Program Files\NSSM\` or any PATH location

**Alternative:** The installer can use `sc.exe` (built into Windows), but this has limitations.

## Installation

---

### Option 1: Using NSSM (Recommended)

#### 1. Install pyWATS Client (if not already installed):

```
powershell pip install pywats-api[client]
```

#### 2. Install the service (run as Administrator):

```
powershell python -m pywats_client install-service
```

This will:

- Create service `pyWATS_Service`
- Set auto-start on boot
- Configure logging to `C:\ProgramData\Virinco\pyWATS\logs\`
- Use default configuration

#### 1. Start the service:

```
powershell net start pyWATS_Service
```

or

```
powershell nssm start pyWATS_Service
```

## Option 2: Native Windows Service (Recommended for Enterprise)

The native Windows service uses pywin32 and provides:

- **Appears in Task Manager** → Services tab
- **Automatic restart on failure** (5s/5s/30s delays)
- **Delayed auto-start** (waits for network services)
- **Windows Event Log integration** (events in Event Viewer)

```
Install native service (run as Administrator)
python -m pywats_client install-service --native

Start the service
net start pyWATS_Service
```

### Features automatically configured:

- Service recovery: restarts after 5s on first two failures, 30s thereafter
- Delayed start: waits for network to be ready before starting
- Event logging: service events written to Windows Event Log

## Option 3: Using sc.exe (Fallback)

If NSSM is not available:

```
python -m pywats_client install-service --use-sc
net start pyWATS_Service
```

**Note:** sc.exe has limitations (no automatic log rotation, limited crash recovery).

## Multi-Instance Installation

For multi-station setups where you need multiple services (one per test station):

```
Install service for Station A
python -m pywats_client install-service --instance-id station_a --config
"C:\ProgramData\Virinco\pyWATS\config_station_a.json"

Install service for Station B
python -m pywats_client install-service --instance-id station_b --config
"C:\ProgramData\Virinco\pyWATS\config_station_b.json"
```

Each instance will have:

- Service name: pyWATS\_Service\_station\_a , pyWATS\_Service\_station\_b

- Separate logs: `pyWATS_Service_station_a.log`, `pyWATS_Service_station_b.log`
- Independent configuration files

## Service Management

### Check Service Status

```
Using sc.exe
sc query pyWATS_Service

Using NSSM
nssm status pyWATS_Service

Using services.msc GUI
services.msc
```

### Start/Stop/Restart

```
Start
net start pyWATS_Service

Stop
net stop pyWATS_Service

Restart
net stop pyWATS_Service && net start pyWATS_Service

Or with NSSM
nssm restart pyWATS_Service
```

### View Logs

Logs are written to `C:\ProgramData\Virinco\pyWATS\logs\`:

- `pyWATS_Service.log` - Standard output
- `pyWATS_Service_error.log` - Error output

```
View latest logs
Get-Content "C:\ProgramData\Virinco\pyWATS\logs\pyWATS_Service.log" -Tail 50

Monitor live
Get-Content "C:\ProgramData\Virinco\pyWATS\logs\pyWATS_Service.log" -Wait
```

## Uninstall Service

```
Stop and remove
python -m pywats_client uninstall-service

For specific instance
python -m pywats_client uninstall-service --instance-id station_a
```

## Configuration

### Default Configuration

The service uses configuration from:

- Default: C:\ProgramData\Virinco\pyWATS\config.json
- Custom: Specify with --config during installation

### Changing Configuration

#### Option 1: Using GUI

1. Run the pyWATS Client GUI
2. It will discover the running service
3. Make configuration changes in the GUI
4. Changes are sent via IPC to the service

#### Option 2: Edit config.json

1. Stop the service: net stop pyWATS\_Service
2. Edit: C:\ProgramData\Virinco\pyWATS\config.json
3. Start the service: net start pyWATS\_Service

#### Option 3: Reinstall with new config

```
python -m pywats_client uninstall-service
python -m pywats_client install-service --config "C:\path\to\new\config.json"
net start pyWATS_Service
```

## Troubleshooting

### Service Won't Start

#### 1. Check logs:

```
powershell Get-Content "C:\ProgramData\Virinco\pyWATS\logs\pyWATS_Service_error.log"
```

## 2. Test service command manually:

```
powershell python -m pywats_client service --instance-id default
```

This runs the service in foreground mode for debugging.

## 1. Verify Python path:

```
powershell where python
```

NSSM uses the Python executable from your PATH. Make sure it's correct.

## Permission Errors

The service runs under the SYSTEM account by default. If you need access to network shares or user-specific resources:

```
Change service account (NSSM)
nssm set pyWATS_Service ObjectName "DOMAIN\Username" "Password"

Or use sc.exe
sc config pyWATS_Service obj= "DOMAIN\Username" password= "Password"
```

## Service Crashes

NSSM automatically restarts crashed services. Check logs for crash details:

```
Get-Content "C:\ProgramData\Virinco\pyWATS\logs\pyWATS_Service_error.log" -Tail 100
```

To disable auto-restart (for debugging):

```
nssm set pyWATS_Service AppExit Default Exit
```

## Multiple Instances Conflict

If you see errors about ports or IPC endpoints already in use:

1. Each instance needs a unique --instance-id

2. Check running services:

```
powershell sc query type= service state= all | findstr "pyWATS"
```

3. Stop conflicting instances:

```
powershell net stop pyWATS_Service net stop pyWATS_Service_station_a
```

# Silent Installation (IT Deployment)

For scripted deployment via GPO, SCCM, or automation tools:

## Basic Silent Install

```
Install silently with native service
python -m pywats_client install-service --native --silent

Check exit code
if ($LASTEXITCODE -ne 0) {
 Write-Error "Installation failed with exit code $LASTEXITCODE"
 exit 1
}
```

## Silent Install with Configuration

```
python -m pywats_client install-service --native --silent `
 --server-url "https://wats.company.com" `
 --api-token "your-api-token" `
 --watch-folder "C:\TestReports"
```

## Exit Codes

Code	Meaning
0	Success
1	General error
2	Missing requirements (Python version, pywin32)
10	Service already installed
11	Service not installed (uninstall)
14	Permission denied (need Administrator)
41	Server unreachable

## Example Deployment Script

```
deploy_pywats.ps1 - Silent deployment script

param(
 [string]$ServerUrl = "https://wats.company.com",
 [string]$ApiToken,
 [string]$WatchFolder = "C:\TestReports"
)

Ensure admin privileges
if (-NOT ([Security.Principal.WindowsPrincipal]
[Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([Security.Principal.WindowsBuiltInRole]
"Administrator")) {
 Write-Error "Administrator privileges required"
 exit 14
}

Install Python package (if needed)
pip install pywats-api[client] --quiet

Install service
python -m pywats_client install-service --native --silent `

--server-url $ServerUrl `

--api-token $ApiToken `

--watch-folder $WatchFolder

if ($LASTEXITCODE -ne 0) {
 Write-Error "Service installation failed: exit code $LASTEXITCODE"
 exit $LASTEXITCODE
}

Start service
net start pyWATS_Service
Write-Host "pyWATS Service installed and started successfully"
```

## Event Log

When using `--native`, the service writes to Windows Event Log:

```
View pyWATS events in Event Viewer
Get-EventLog -LogName Application -Source "pyWATS" -Newest 20
```

Events include:

- Service installation/uninstallation
- Service start/stop
- Errors and warnings

# Advanced Configuration

---

## Custom Service Name

```
Edit WindowsServiceInstaller.SERVICE_NAME in:
src/pywats_client/control/windows_service.py
```

## Environment Variables

```
Set environment variables for the service
nssm set pyWATS_Service AppEnvironmentExtra PYTHONPATH=C:\custom\path
```

## Delayed Start

```
Start service 2 minutes after boot
sc config pyWATS_Service start= delayed-auto
```

## GUI Discovery

---

When you open the pyWATS Client GUI:

- 1. Discovery:** GUI scans for running service instances
- 2. Instance Selector:** Shows all discovered services
- 3. Connect:** Select an instance to view/configure
- 4. Status:** Live status updates via IPC

The GUI never auto-starts services - they must be started separately (manually or via Windows Service).

## Comparison with Manual Start

---

Method	Auto-Start	Survives Reboot	Crash Recovery	Service Management
<b>Windows Service</b>	✓	✓	✓ (NSSM)	services.msc, sc.exe
<b>Manual (GUI)</b>	X	X	X	Task Manager
<b>Task Scheduler</b>	✓	✓	X	taskschd.msc
<b>Startup Folder</b>	✓	✓	X	Manual

**Recommendation:** Use Windows Service for production environments.

## See Also

---

- Getting Started - Basic client usage
  - Client Installation - Installation guide
  - NSSM Documentation - Full NSSM options
- 

*Source: docs/installation/linux-service.md*

# Linux systemd Service Installation

---

This guide explains how to install pyWATS Client as a Linux systemd service that auto-starts on system boot.

## Overview

---

The pyWATS Client can run as a systemd service in the background, automatically starting when Linux boots. This is the recommended setup for production environments.

### System Compatibility:

- Ubuntu 16.04+ / Debian 8+
- RHEL/CentOS 7+
- Fedora 15+
- Any systemd-based Linux distribution

### Folder Structure:

- **System-wide:** /var/lib/pywats/ (configuration, logs, queues)
- **User-specific:** ~/.config/pywats\_client/
- **Service files:** /etc/systemd/system/

## Prerequisites

---

### Required

- Linux with systemd (check: `systemctl --version`)
- Python 3.10 or later
- Root privileges (for system-wide installation)

## Check systemd

```
Verify systemd is running
systemctl --version

Should show systemd version (e.g., systemd 245)
```

## Installation

---

### System-Wide Installation (Recommended)

Installs service that runs at boot, before user login.

```
Install pyWATS Client
pip install pywats-api[client]

Install as systemd service (requires sudo)
sudo python -m pywats_client install-service

Start the service
sudo systemctl start pywats-service

Check status
sudo systemctl status pywats-service
```

This creates:

- Service: pywats-service.service
- Auto-start: Enabled
- Logs: journalctl -u pywats-service
- Data directory: /var/lib/pywats/

### User-Level Installation

Installs service that runs when specific user logs in.

```
Install pyWATS Client
pip install pywats-api[client]

Install as user service (specify your username)
sudo python -m pywats_client install-service --user $USER

Start the service
sudo systemctl start pywats-service

Check status
systemctl status pywats-service
```

This creates:

- Service runs as your user account
- Data directory: `~/.config/pywats_client/`
- User-specific permissions

## Multi-Instance Installation

---

For multi-station setups where you need multiple services (one per test station):

```
Create separate config files
sudo mkdir -p /var/lib/pywats
sudo cp config.json /var/lib/pywats/config_station_a.json
sudo cp config.json /var/lib/pywats/config_station_b.json

Install service for Station A
sudo python -m pywats_client install-service \
 --instance-id station_a \
 --config /var/lib/pywats/config_station_a.json \
 --user pywats

Install service for Station B
sudo python -m pywats_client install-service \
 --instance-id station_b \
 --config /var/lib/pywats/config_station_b.json \
 --user pywats

Start both services
sudo systemctl start pywats-service@station_a
sudo systemctl start pywats-service@station_b
```

Each instance will have:

- Service name: `pywats-service@station_a.service` , `pywats-service@station_b.service`
- Separate configurations
- Independent logging

# Service Management

## Check Service Status

```
Status
sudo systemctl status pywats-service

Check if enabled
systemctl is-enabled pywats-service

Check if running
systemctl is-active pywats-service
```

## Start/Stop/Restart

```
Start
sudo systemctl start pywats-service

Stop
sudo systemctl stop pywats-service

Restart
sudo systemctl restart pywats-service

Reload configuration (without restart)
sudo systemctl reload pywats-service
```

## Enable/Disable Auto-Start

```
Enable (auto-start on boot)
sudo systemctl enable pywats-service

Disable (don't auto-start)
sudo systemctl disable pywats-service

Check status
systemctl is-enabled pywats-service
```

## View Logs

systemd logs all service output to the journal:

```
View all logs
sudo journalctl -u pywats-service

Follow logs (live)
sudo journalctl -u pywats-service -f

Last 50 lines
sudo journalctl -u pywats-service -n 50

Since last boot
sudo journalctl -u pywats-service -b

Last hour
sudo journalctl -u pywats-service --since "1 hour ago"

With timestamps
sudo journalctl -u pywats-service -o short-iso
```

## Uninstall Service

```
Stop and remove
sudo python -m pywats_client uninstall-service

For specific instance
sudo python -m pywats_client uninstall-service --instance-id station_a

Verify removal
systemctl list-units | grep pywats
```

## Silent Installation (IT Deployment)

For automated deployment via Ansible, Puppet, Chef, or shell scripts, use silent mode with exit codes.

## Silent Mode Parameters

Parameter	Description
--silent	Suppress all output (exit codes only)
--server-url URL	Pre-configure WATS server URL
--api-token TOKEN	Pre-configure API token
--watch-folder PATH	Pre-configure watch folder path
--skip-preflight	Skip connectivity checks
--instance-id ID	Create named instance
--user USERNAME	Run service as specific user

## Exit Codes

Code	Meaning
0	Success
1	General error
2	Missing requirements (systemd not available)
10	Service already installed
11	Service not installed (uninstall)
14	Permission denied (not running as root)
20-22	Configuration errors
30	Server unreachable

## Bash Deployment Script

```
#!/bin/bash
deploy_pywats.sh - Silent deployment for Ubuntu/Debian

set -e

WATS_SERVER="https://your-wats-server.com"
API_TOKEN="your-api-token"
WATCH_FOLDER="/data/reports"
SERVICE_USER="pywats"

Create service user if not exists
if ! id "$SERVICE_USER" &>/dev/null; then
 useradd -r -m -d /var/lib/pywats -s /bin/false "$SERVICE_USER"
fi

Create directories
mkdir -p /var/lib/pywats /var/log/pywats "$WATCH_FOLDER"
chown -R "$SERVICE_USER:$SERVICE_USER" /var/lib/pywats /var/log/pywats

Install Python package
pip3 install pywats-api[client] --quiet

Install service silently
python3 -m pywats_client install-service \
 --silent \
 --server-url "$WATS_SERVER" \
 --api-token "$API_TOKEN" \
 --watch-folder "$WATCH_FOLDER" \
 --user "$SERVICE_USER"

EXIT_CODE=$?

case $EXIT_CODE in
 0) echo "Installation successful"
 systemctl start pywats-service
 ;;
 10) echo "Already installed, restarting..."
 systemctl restart pywats-service
 ;;
 14) echo "ERROR: Must run as root"
 exit 1
 ;;
 *) echo "ERROR: Installation failed (code $EXIT_CODE)"
 exit 1
 ;;
esac

Verify service is running
systemctl is-active pywats-service
```

## Ansible Playbook

```
deploy_pywats.yml

- name: Deploy pyWATS Client
 hosts: test_stations
 become: yes
 vars:
 wats_server: "https://wats.example.com"
 api_token: "{{ vault_api_token }}"
 watch_folder: "/data/reports"
 service_user: "pywats"

 tasks:
 - name: Create service user
 user:
 name: "{{ service_user }}"
 system: yes
 home: /var/lib/pywats
 shell: /bin/false
 create_home: yes

 - name: Create directories
 file:
 path: "{{ item }}"
 state: directory
 owner: "{{ service_user }}"
 group: "{{ service_user }}"
 mode: '0755'
 loop:
 - /var/lib/pywats
 - /var/log/pywats
 - "{{ watch_folder }}"

 - name: Install pyWATS
 pip:
 name: pywats-api[client]
 state: latest

 - name: Install service
 command: >
 python3 -m pywats_client install-service
 --silent
 --server-url {{ wats_server }}
 --api-token {{ api_token }}
 --watch-folder {{ watch_folder }}
 --user {{ service_user }}
 register: install_result
 changed_when: install_result.rc == 0
 failed_when: install_result.rc not in [0, 10]

 - name: Start service
 systemd:
 name: pywats-service
```

```
state: started
enabled: yes
```

## Query Installation Status

```
Check if service is installed
python3 -m pywats_client status --instance-id default
Returns exit code 0 if installed, 11 if not

Get service status
systemctl is-active pywats-service && echo "Running" || echo "Stopped"
```

## Configuration

### Default Configuration

The service uses configuration from:

- System-wide: /var/lib/pywats/config.json
- User-specific: ~/.config/pywats\_client/config.json
- Custom: Specify with --config during installation

### Changing Configuration

#### Option 1: Using GUI

1. Run the pyWATS Client GUI
2. It will discover the running service
3. Make configuration changes in the GUI
4. Changes are sent via IPC to the service

#### Option 2: Edit config.json

```
Stop the service
sudo systemctl stop pywats-service

Edit configuration
sudo nano /var/lib/pywats/config.json

Start the service
sudo systemctl start pywats-service

Verify
sudo journalctl -u pywats-service -n 20
```

#### Option 3: Reinstall with new config

```
sudo python -m pywats_client uninstall-service
sudo python -m pywats_client install-service --config /path/to/new/config.json
sudo systemctl start pywats-service
```

## Troubleshooting

### Service Won't Start

#### 1. Check logs:

```
bash sudo journalctl -u pywats-service -n 100 --no-pager
```

#### 2. Check service file:

```
bash sudo systemctl cat pywats-service
```

#### 3. Test service command manually:

```
bash # Run in foreground for debugging python -m pywats_client service --instance-id default
```

#### 4. Verify Python path:

```
bash which python python --version
```

#### 5. Check permissions:

```
bash ls -la /var/lib/pywats/ sudo chown -R pywats:pywats /var/lib/pywats/
```

## Permission Errors

If the service can't access files:

```
Create dedicated user
sudo useradd -r -s /bin/false pywats

Set ownership
sudo chown -R pywats:pywats /var/lib/pywats/

Reinstall with user
sudo python -m pywats_client uninstall-service
sudo python -m pywats_client install-service --user pywats
```

## Service Keeps Restarting

The service is configured to automatically restart on failure. Check why it's failing:

```
View recent crashes
sudo journalctl -u pywats-service --since "10 minutes ago"

Disable auto-restart temporarily
sudo systemctl edit pywats-service
Add:
[Service]
Restart=no

sudo systemctl daemon-reload
sudo systemctl restart pywats-service
```

## Port Already in Use

If you see "Address already in use" errors:

```
Check what's using the port
sudo netstat -tlnp | grep :8765

Or with ss
sudo ss -tlnp | grep :8765

Kill the conflicting process or change port in config
```

## Multiple Instances Conflict

If you see errors about IPC endpoints already in use:

```
List all pyWATS services
systemctl list-units | grep pywats

Check each one
sudo systemctl status pywats-service
sudo systemctl status pywats-service@station_a

Ensure each has unique instance-id
```

# Advanced Configuration

## Custom User and Group

```
Create dedicated user
sudo useradd -r -m -d /var/lib/pywats -s /bin/false pywats

Install with custom user
sudo python -m pywats_client install-service --user pywats
```

## Environment Variables

Edit the service file:

```
sudo systemctl edit pywats-service --full
```

Add environment variables:

```
[Service]
Environment="PYTHONPATH=/custom/path"
Environment="PYWATS_LOG_LEVEL=DEBUG"
```

Then reload:

```
sudo systemctl daemon-reload
sudo systemctl restart pywats-service
```

## Resource Limits

The service unit file includes production-hardened defaults:

```
[Service]
Memory and CPU limits
MemoryMax=512M
CPUQuota=80%
LimitNOFILE=65535
LimitNPROC=4096

Watchdog (service health monitoring)
Type=notify
WatchdogSec=60s

Security hardening
NoNewPrivileges=true
PrivateTmp=true
ProtectSystem=strict
ProtectHome=read-only
PrivateDevices=true
ProtectKernelTunables=true
ProtectKernelModules=true
ProtectControlGroups=true

Capability restrictions
CapabilityBoundingSet=CAP_NET_BIND_SERVICE

System call filtering
SystemCallArchitectures=native
SystemCallFilter=@system-service
```

To customize limits, override with:

## Network Dependencies

If pyWATS needs specific network services:

```
sudo systemctl edit pywats-service --full
```

Add dependencies:

```
[Unit]
After=network-online.target postgresql.service
Requires=network-online.target
```

## GUI Discovery

When you open the pyWATS Client GUI on Linux:

1. **Discovery:** GUI scans for running service instances via IPC
2. **Instance Selector:** Shows all discovered services

3. **Connect**: Select an instance to view/configure

4. **Status**: Live status updates via IPC

The GUI never auto-starts services - they must be started separately via systemd.

## Comparison with Other Methods

---

Method	Auto-Start	Survives Reboot	Crash Recovery	Service Management
<b>systemd</b>	✓	✓	✓	systemctl, journalctl
<b>Manual (GUI)</b>	X	X	X	Terminal
<b>cron @reboot</b>	✓	✓	X	crontab
<b>init.d script</b>	✓	✓	X	service command

**Recommendation:** Use systemd for production environments.

## Ubuntu Specific Notes

---

### Ubuntu 22.04 LTS (Jammy)

Fully supported. systemd 249.

```
sudo apt update
sudo apt install python3 python3-pip
pip3 install pywats-api[client]
sudo python3 -m pywats_client install-service
```

### Ubuntu 20.04 LTS (Focal)

Fully supported. systemd 245.

### Ubuntu 18.04 LTS (Bionic)

Supported. systemd 237.

## See Also

---

- Getting Started - Basic client usage

- Client Installation - Installation guide
  - systemd documentation
- 

Source: [docs/installation/macoss-service.md](#)

# macOS launchd Service Installation

---

This guide explains how to install pyWATS Client as a macOS launchd service that auto-starts on system boot.

## Overview

---

The pyWATS Client can run as a launchd daemon/agent in the background, automatically starting when macOS boots or when you log in.

### Service Types:

- **Launch Daemon:** Starts at boot (system-wide, requires sudo)
- **Launch Agent:** Starts at login (user-specific, no sudo required)

### Folder Structure:

- **Daemons:** /Library/LaunchDaemons/ (system-wide)
- **Agents:** /Library/LaunchAgents/ (user-level)
- **Data (Daemon):** /var/lib/pywats/
- **Data (Agent):** ~/.config/pywats\_client/
- **Logs (Daemon):** /var/log/pywats/
- **Logs (Agent):** ~/Library/Logs/pyWATS/

## Prerequisites

---

### Required

- macOS 10.10 (Yosemite) or later
- Python 3.10 or later
- Administrator privileges (for Launch Daemon)

# Installation

---

## Option 1: Launch Daemon (System-Wide, Recommended)

Runs at boot, before any user logs in. Best for production test stations.

```
Install pyWATS Client
pip3 install pywats-api[client]

Install as Launch Daemon (requires sudo)
sudo python3 -m pywats_client install-service

Service starts automatically after installation
Or start manually:
sudo launchctl start com.virinco.pywats.service
```

This creates:

- Plist: /Library/LaunchDaemons/com.virinco.pywats.service.plist
- Auto-start: Enabled (runs at boot)
- Logs: /var/log/pywats/pywats-service.log
- Data: /var/lib/pywats/

## Option 2: Launch Agent (User-Level)

Runs when you log in. Good for development.

```
Install pyWATS Client
pip3 install pywats-api[client]

Install as Launch Agent (no sudo needed)
python3 -m pywats_client install-service --user-agent

Service starts automatically at login
Or start manually:
launchctl start com.virinco.pywats.service
```

This creates:

- Plist: /Library/LaunchAgents/com.virinco.pywats.service.plist
- Auto-start: At login
- Logs: ~/Library/Logs/pyWATS/pywats-service.log
- Data: ~/.config/pywats\_client/

## Multi-Instance Installation

---

For multi-station setups where you need multiple services (one per test station):

```

Create config files
sudo mkdir -p /var/lib/pywats
sudo cp config.json /var/lib/pywats/config_station_a.json
sudo cp config.json /var/lib/pywats/config_station_b.json

Install service for Station A
sudo python3 -m pywats_client install-service \
--instance-id station_a \
--config /var/lib/pywats/config_station_a.json

Install service for Station B
sudo python3 -m pywats_client install-service \
--instance-id station_b \
--config /var/lib/pywats/config_station_b.json

```

Each instance will have:

- Plist: com.virinco.pywats.service.station\_a.plist , com.virinco.pywats.service.station\_b.plist
- Separate configurations
- Independent logging

## Service Management

---

### Check Service Status

```

List all launchd services (system)
sudo launchctl list | grep pywats

List user services
launchctl list | grep pywats

Check specific service
sudo launchctl list com.virinco.pywats.service

```

### Start/Stop

#### Launch Daemon (system-wide):

```

Start
sudo launchctl start com.virinco.pywats.service

Stop
sudo launchctl stop com.virinco.pywats.service

Note: Service will auto-restart if killed
To prevent restart, unload the plist
sudo launchctl unload /Library/LaunchDaemons/com.virinco.pywats.service.plist

```

## **Launch Agent (user-level):**

```
Start
launchctl start com.virinco.pywats.service

Stop
launchctl stop com.virinco.pywats.service

Unload
launchctl unload /Library/LaunchAgents/com.virinco.pywats.service.plist
```

## **View Logs**

### **Launch Daemon logs:**

```
View logs
sudo tail -f /var/log/pywats/pywats-service.log

View errors
sudo tail -f /var/log/pywats/pywats-service-error.log

View with Console.app
open -a Console /var/log/pywats/
```

### **Launch Agent logs:**

```
View logs
tail -f ~/Library/Logs/pyWATS/pywats-service.log

View errors
tail -f ~/Library/Logs/pyWATS/pywats-service-error.log

View with Console.app
open -a Console ~/Library/Logs/pyWATS/
```

## **Uninstall Service**

### **Launch Daemon:**

```
Stop and remove
sudo python3 -m pywats_client uninstall-service

For specific instance
sudo python3 -m pywats_client uninstall-service --instance-id station_a

Verify removal
sudo launchctl list | grep pywats
```

## Launch Agent:

```
Stop and remove
python3 -m pywats_client uninstall-service --user-agent

Verify removal
launchctl list | grep pywats
```

# Configuration

## Default Configuration

The service uses configuration from:

- Daemon: /var/lib/pywats/config.json
- Agent: ~/.config/pywats\_client/config.json
- Custom: Specify with --config during installation

## Changing Configuration

### Option 1: Using GUI

1. Run the pyWATS Client GUI
2. It will discover the running service
3. Make configuration changes in the GUI
4. Changes are sent via IPC to the service

### Option 2: Edit config.json

```
Stop the service
sudo launchctl stop com.virinco.pywats.service

Edit configuration (Daemon)
sudo nano /var/lib/pywats/config.json

Or for Agent
nano ~/.config/pywats_client/config.json

Restart the service
sudo launchctl start com.virinco.pywats.service
```

### Option 3: Reinstall with new config

```
sudo python3 -m pywats_client uninstall-service
sudo python3 -m pywats_client install-service --config /path/to/new/config.json
```

# Troubleshooting

## Service Won't Start

### 1. Check if plist is loaded:

```
bash sudo launchctl list | grep pywats
```

### 2. Check logs:

```
bash sudo tail -100 /var/log/pywats/pywats-service-error.log
```

### 3. Test service command manually:

```
bash # Run in foreground for debugging python3 -m pywats_client service --instance-id default
```

### 4. Verify Python path:

```
bash which python3 python3 --version
```

### 5. Check plist syntax:

```
bash plutil /Library/LaunchDaemons/com.virinco.pywats.service.plist
```

## Permission Errors

If the service can't access files:

```
Check ownership (Daemon)
ls -la /var/lib/pywats/

Fix permissions
sudo chown -R root:wheel /var/lib/pywats/
sudo chmod -R 755 /var/lib/pywats/

For Agent
ls -la ~/.config/pywats_client/
```

## Service Keeps Restarting

The service is configured to auto-restart on failure. Check why it's failing:

```
View recent errors
sudo tail -100 /var/log/pywats/pywats-service-error.log

Check system logs
log show --predicate 'subsystem == "com.apple.launchd"' --last 10m | grep pywats

Disable auto-restart temporarily
sudo launchctl unload /Library/LaunchDaemons/com.virinco.pywats.service.plist
```

## Port Already in Use

If you see "Address already in use" errors:

```
Check what's using the port
sudo lsof -i :8765

Kill the conflicting process or change port in config
```

## Service Doesn't Auto-Start at Boot

### 1. Verify plist location:

```
bash ls -l /Library/LaunchDaemons/com.virinco.pywats.service.plist
```

### 2. Check RunAtLoad:

```
bash plutil -p /Library/LaunchDaemons/com.virinco.pywats.service.plist | grep RunAtLoad #
Should show: "RunAtLoad" => 1
```

### 3. Reload plist:

```
bash sudo launchctl unload /Library/LaunchDaemons/com.virinco.pywats.service.plist sudo
launchctl load /Library/LaunchDaemons/com.virinco.pywats.service.plist
```

## Advanced Configuration

### Custom Environment Variables

Edit the plist file:

```
sudo nano /Library/LaunchDaemons/com.virinco.pywats.service.plist
```

Add environment variables:

```
<key>EnvironmentVariables</key>
<dict>
 <key>PYTHONPATH</key>
 <string>/custom/path</string>
 <key>PYWATS_LOG_LEVEL</key>
 <string>DEBUG</string>
</dict>
```

Then reload:

```
sudo launchctl unload /Library/LaunchDaemons/com.virinco.pywats.service.plist
sudo launchctl load /Library/LaunchDaemons/com.virinco.pywats.service.plist
```

## Run on Schedule

Instead of continuous running, run periodically:

```
<key>StartCalendarInterval</key>
<dict>
 <key>Hour</key>
 <integer>8</integer>
 <key>Minute</key>
 <integer>0</integer>
</dict>
```

## Resource Limits

Add limits to prevent runaway processes:

```
<key>SoftResourceLimits</key>
<dict>
 <key>NumberOfFiles</key>
 <integer>1024</integer>
</dict>

<key>HardResourceLimits</key>
<dict>
 <key>NumberOfFiles</key>
 <integer>2048</integer>
</dict>
```

## Network Dependency

Ensure network is available before starting:

The default plist already includes this via `StandardOutPath` and `StandardErrorPath`, which ensures the filesystem is ready.

## GUI Discovery

When you open the pyWATS Client GUI on macOS:

1. **Discovery:** GUI scans for running service instances via IPC
2. **Instance Selector:** Shows all discovered services
3. **Connect:** Select an instance to view/configure
4. **Status:** Live status updates via IPC

The GUI never auto-starts services - they must be started separately via launchd.

## Comparison with Other Methods

---

Method	Auto-Start	Survives Reboot	Crash Recovery	Service Management
<b>launchd Daemon</b>	✓ (boot)	✓	✓	launchctl
<b>launchd Agent</b>	✓ (login)	✓	✓	launchctl
<b>Manual (GUI)</b>	X	X	X	Terminal
<b>Login Item</b>	✓ (login)	✓	X	System Preferences
<b>cron @reboot</b>	✓	✓	X	crontab

**Recommendation:** Use Launch Daemon for production test stations.

## macOS Version Notes

---

### macOS 13+ (Ventura)

Fully supported. May require additional privacy permissions.

```
Grant Full Disk Access if needed
System Settings → Privacy & Security → Full Disk Access
```

### macOS 12 (Monterey)

Fully supported.

### macOS 11 (Big Sur)

Fully supported.

### macOS 10.15 (Catalina)

Supported. May prompt for security approval on first run.

## See Also

---

- Getting Started - Basic client usage
- Client Installation - Installation guide

- Apple launchd documentation

# Architecture

Source: [docs/architecture.md](#)

## pyWATS System Architecture

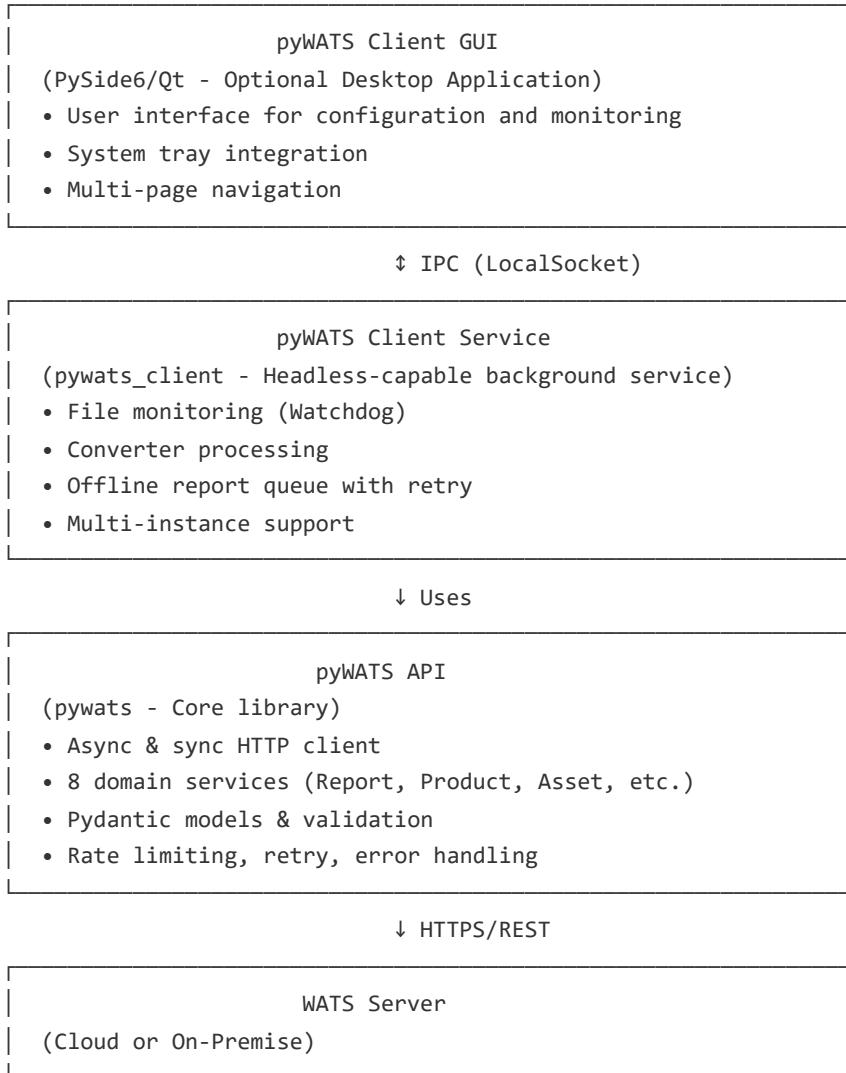
**Version:** 1.3.0

**Last Updated:** January 2026

**Audience:** Developers, integrators, contributors

## Overview

pyWATS is a **three-layer Python system** for connecting manufacturing test stations to WATS (Web-based Automatic Test System) servers. It provides both a programmatic API for test automation and a client application for test station deployment.



## Key Features:

- **Dual API modes:** Async ( AsyncWATS ) for high-performance concurrent operations, Sync ( pyWATS ) for simple scripting
- **Headless operation:** Client runs without GUI on servers, Raspberry Pi, embedded systems
- **Offline resilience:** Queue and retry reports when network is unavailable
- **Multi-instance:** Run multiple clients on same machine with separate configurations
- **Extensible:** Custom converters, custom domains, plugin architecture

## Table of Contents

1. System Layers
2. pyWATS API Layer
3. pyWATS Client Layer
4. pyWATS GUI Layer
5. Configuration Management

6. Async vs Sync Usage
  7. Extension Points
  8. Deployment Modes
  9. Dependencies
- 

## System Layers

---

### Layer 1: pyWATS API ( `src/pywats/` )

**Purpose:** Foundation library for REST API communication with WATS servers

**Responsibility:**

- HTTP/HTTPS communication with authentication
- Domain-specific business logic (8 domains)
- Pydantic model validation
- Rate limiting and retry logic
- Error handling and responses

**Used by:** Test automation scripts, client service, custom integrations

**Entry point:** `pyWATS` (sync) or `AsyncWATS` (async)

### Layer 2: pyWATS Client Service ( `src/pywats_client/` )

**Purpose:** Background service for test station automation

**Responsibility:**

- File system monitoring (Watchdog)
- Converter execution (file → report transformation)
- Offline queue management with persistence
- Connection health monitoring
- Process/product data synchronization
- Multi-instance coordination

**Modes:** Headless daemon, Windows/Linux/macOS service, foreground process

**Entry point:** `python -m pywats_client service`

### Layer 3: pyWATS GUI ( `src/pywats_client/gui/` )

**Purpose:** Optional desktop application for client configuration

**Responsibility:**

- User-friendly configuration interface

- Visual status monitoring
- Log viewing and troubleshooting
- System tray integration

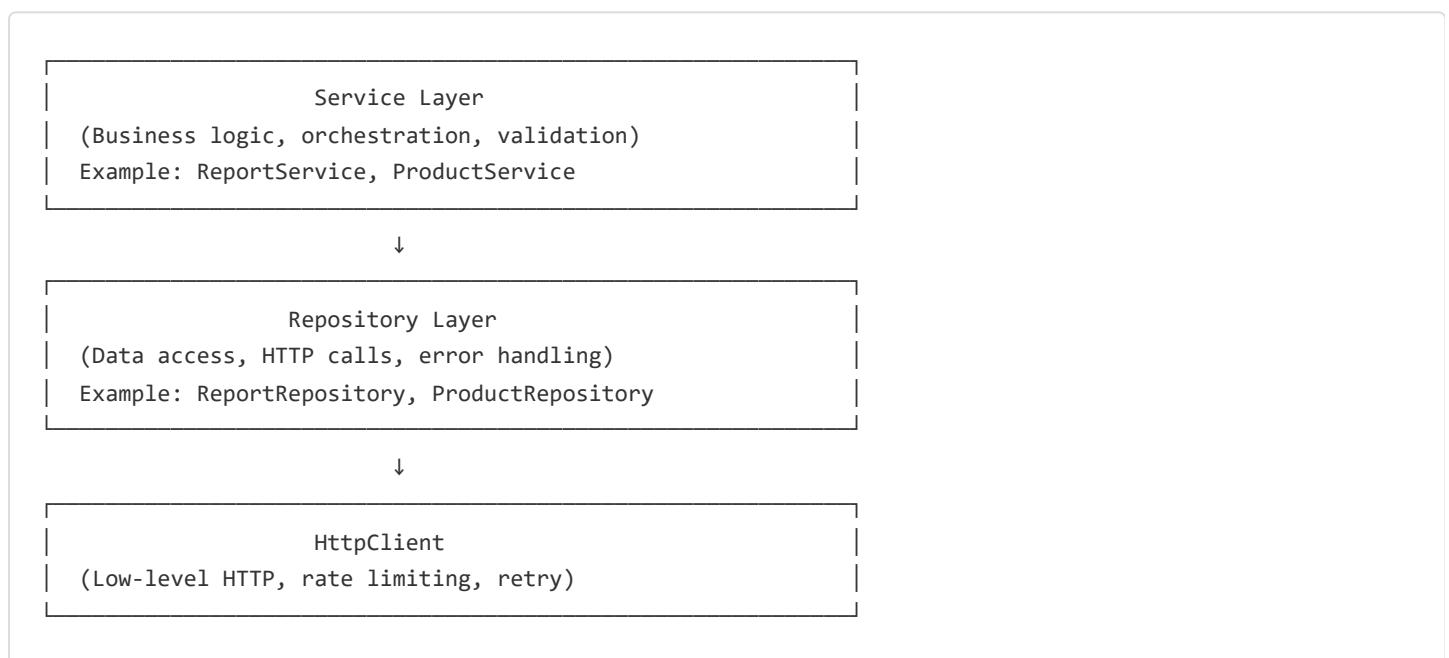
**Technology:** PySide6 (Qt6 for Python)

**Entry point:** python -m pywats\_client gui

## pyWATS API Layer

### Architecture Pattern

Each domain follows a **three-layer pattern**:



### Core Components

#### HttpClient ( core/client.py )

##### Both sync and async implementations:

- HttpClient - Synchronous (uses requests )
- AsyncHttpClient - Asynchronous (uses httpx )

##### Features:

- Basic authentication (username + API token)
- Automatic response wrapping with Response model
- JSON serialization/deserialization
- No exception raising for HTTP errors (repositories handle errors)

- Built-in rate limiting integration
- Automatic retry for transient failures

## Response Model:

```
class Response:
 status_code: int
 data: Any # Parsed JSON
 headers: Dict[str, str]
 raw: bytes

 # Computed properties
 is_success: bool # 2xx
 is_client_error: bool # 4xx
 is_server_error: bool # 5xx
 is_not_found: bool # 404
 error_message: Optional[str]
```

## Rate Limiting ( core/throttle.py )

### Thread-safe sliding window rate limiter:

- Default: 500 requests per 60 seconds
- Configurable via `configure_throttling()`
- Automatic wait when limit reached
- Statistics tracking (total requests, throttle count, wait time)

### Configuration:

```
from pywats.core.throttle import configure_throttling

configure_throttling(
 max_requests=500,
 window_seconds=60,
 enabled=True
)
```

## Retry Logic ( core/retry.py )

### Automatic retry with exponential backoff:

- Default: 3 attempts, 1s base delay, 30s max delay
- Retries on: `ConnectionError`, `TimeoutError`, HTTP 429/5xx
- Respects `Retry-After` header for HTTP 429
- Does NOT retry: POST (not idempotent), HTTP 4xx errors
- Configurable per client instance

### Configuration:

```
from pywats import pyWATS, RetryConfig

config = RetryConfig(
 max_attempts=5,
 base_delay=2.0,
 max_delay=60.0,
 jitter=True
)
api = pyWATS(..., retry_config=config)
```

## Error Handling ( core/exceptions.py , shared/result.py )

### Two error handling modes:

1. **ErrorMode.STRICT** (default) - Raises exceptions
2. `NotFoundError` for HTTP 404
3. `ValidationError` for HTTP 400
4. `AuthenticationError` for HTTP 401
5. `AuthorizationError` for HTTP 403
6. `ServerError` for HTTP 5xx
7. **ErrorMode.LENIENT** - Returns `None` for errors
8. No exceptions raised
9. Repository methods return `None` on 404
10. Suitable for optional data fetching

### Result Pattern (advanced):

```
from pywats.shared.result import Result

result = await api.product.get_product_async("PRODUCT-123")
if result.is_success:
 print(result.value) # Product object
else:
 print(result.error) # ErrorCode.NOT_FOUND
```

## Domain Services

### 8 domains available:

Domain	Service	Purpose
<b>Report</b>	ReportService	Submit test reports (UUT/UUR), query results
<b>Product</b>	ProductService	Manage products, revisions, BOMs
<b>Asset</b>	AssetService	Track equipment, calibration, maintenance
<b>Production</b>	ProductionService	Unit lifecycle, serial numbers, assembly
<b>Analytics</b>	AnalyticsService	Yield analysis, measurements, Cpk
<b>Software</b>	SoftwareService	Package management, distribution
<b>RootCause</b>	RootCauseService	Issue tracking, defect management
<b>Process</b>	ProcessService	Test/repair operations, caching

### Access pattern:

```
from pywats import pyWATS

api = pyWATS(
 base_url="https://company.wats.com",
 token="your-api-token"
)

Access services
api.report.submit_report(report_data)
api.product.get_product("PRODUCT-123")
api.production.allocate_serial_numbers(...)
```

## Station Identity ( core/station.py )

### Station concept for report attribution:

```
from pywats.core.station import Station

station = Station(
 name="STATION-ICT-01", # machineName in reports
 location="Building A",
 purpose="Production" # or "Development"
)

api = pyWATS(..., station=station)
```

### Multi-station support (hub mode):

```
from pywats.core.station import StationRegistry

registry = StationRegistry()
registry.add("ict", Station("ICT-01", ...))
registry.add("fct", Station("FCT-01", ...))

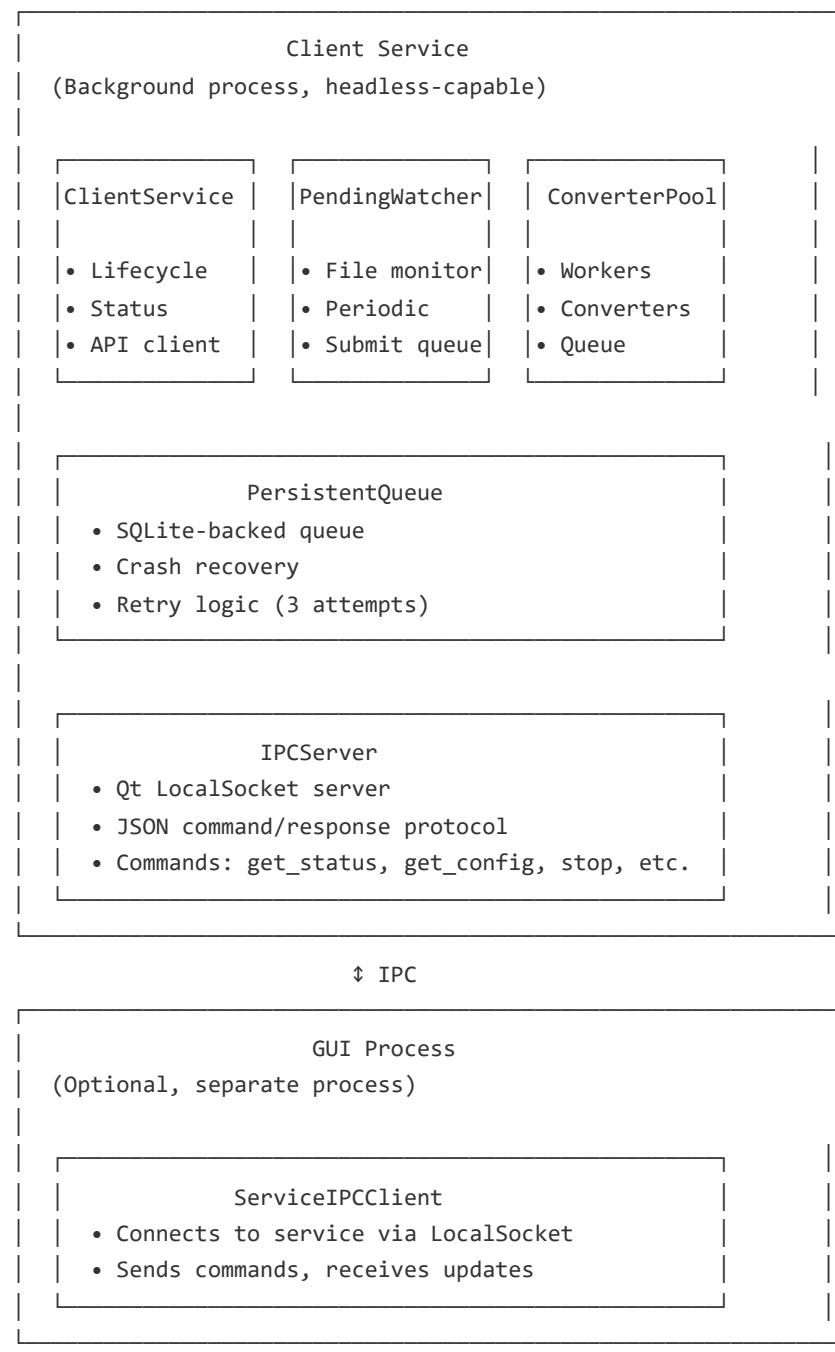
registry.set_active("ict")
current = registry.get_active() # Station("ICT-01", ...)
```

## pyWATS Client Layer

---

### Service Architecture (New in v1.3)

The client has been refactored into **separate service and GUI** with IPC communication:



## Key components:

1. **ClientService** - Main service controller
2. ServiceStatus states: STOPPED, START\_PENDING, RUNNING, STOP\_PENDING, PAUSED
3. Manages lifecycle and coordinates components
4. **PendingWatcher** - Report queue manager
5. Monitors pending reports directory
6. Periodic check (5 minutes) + file system events
7. Submission lock to prevent concurrent uploads
8. **ConverterPool** - Converter worker management

9. 1-50 workers (configurable, default: 10)
10. Manages converter list and pending queue
11. **PersistentQueue** - SQLite-backed queue
12. States: pending, processing, completed, failed
13. Automatic crash recovery (processing → pending on restart)
14. Retry logic with configurable attempts
15. **IPCSERVER** - Inter-process communication
16. Socket name: pyWATS\_Service\_{instance\_id}
17. JSON protocol for commands and responses
18. Service/GUI separation enables headless mode

## Core Services

### File Monitor ( service/file\_monitor.py )

#### File system watching using Watchdog:

- Monitors configured watch folders
- Pattern matching ( \*.csv , \*.xml , etc.)
- Debouncing (waits for file write completion)
- Async event processing
- Callbacks for file events

#### Monitor rules:

```
class MonitorRule:
 path: str # Folder to watch
 converter_type: str # Which converter to use
 recursive: bool # Watch subdirectories
 file_pattern: str # Glob pattern
 delete_after_convert: bool # Auto-cleanup
```

### Converter System ( converters/ )

#### Three converter types:

1. **FileConverter** - One file → one report
2. **FolderConverter** - Entire folder → one report (batch processing)
3. **ScheduledConverter** - Timer-based execution

#### Base classes:

```

from pywats_client.converters import FileConverter, ConverterSource

class MyConverter(FileConverter):
 @property
 def name(self) -> str:
 return "My Test Converter"

 @property
 def file_patterns(self) -> List[str]:
 return [".csv", ".txt"]

 def convert(self, source: ConverterSource, context) -> ConverterResult:
 # Parse file
 data = self._parse_file(source.read_text())

 # Build report using ReportBuilder
 builder = ReportBuilder(
 part_number=data["part_number"],
 serial_number=data["serial_number"]
)

 for test in data["tests"]:
 builder.add_step(
 name=test["name"],
 value=test["value"],
 limits=(test["low"], test["high"])
)

 return ConverterResult.success(builder.to_dict())

```

### **Converter lifecycle:**

1. File created/modified event detected
2. Debounce wait (ensure write complete)
3. Pattern match check
4. Load converter module dynamically
5. Call `convert(source, context)`
6. Submit result to queue or upload directly
7. Post-processing (move/delete/archive based on config)

### **Configuration ( config.py )**

#### **ClientConfig structure:**

- Instance identity (ID, name)
- Server connection (URL, token, username)
- Station identification
- Serial number handler settings
- Proxy configuration
- Sync settings (interval, enabled)
- Offline queue settings
- Converter configurations

- GUI tab visibility
- Logging configuration

### **Encryption:**

- API tokens encrypted with machine-specific key
- Uses `cryptography.fernet` (symmetric encryption)
- Machine ID from system (Windows Registry, `/etc/machine-id`, `IOPlatformUUID`)
- Tokens cannot be moved between machines

### **Configuration locations:**

```
Windows: %APPDATA%\pyWATS_Client\
Linux: ~/.config/pywats_client/
macOS: ~/Library/Application Support/pywats_client/
```

#### Structure:

```
config.json # Default instance
config_{instance_id}.json # Named instances
client.log # Application log
cache/
 processes_cache.json # Synced reference data
reports/
 pending/ # Queued for upload
 processing/ # Currently uploading
 completed/ # Successfully uploaded
 failed/ # Max retries exceeded
```

### **Instance Management ( `service/instance_manager.py` )**

#### **Multi-instance support:**

- File-based locking ( `.lock` files in temp directory)
- Lock contains: instance\_id, name, PID, started timestamp
- Stale lock detection (checks if PID still running)
- Each instance has separate:
  - Configuration file
  - Queue directory
  - Log file
  - IPC socket name

**Use case:** Multiple test processes on same machine (e.g., ICT + FCT + EOL on one PC)

# pyWATS GUI Layer

## Application Structure

```
src/pywats_client/gui/
├── app.py # Main entry point, QApplication setup
├── login_window.py # Pre-authentication dialog
├── main_window.py # Main application window
├── settings_dialog.py # Settings configuration
├── styles.py # Dark theme stylesheet
└── pages/ # Stacked widget pages
 ├── base.py # BasePage abstract class
 ├── setup.py # Initial configuration
 ├── connection.py # Server connection status
 ├── converters.py # Converter management
 ├── sn_handler.py # Serial number handling
 ├── software.py # Software distribution
 ├── about.py # Version info
 └── log.py # Application logs
└── widgets/ # Reusable UI components
```

## Communication with Service

### IPC Protocol (Qt LocalSocket):

```
GUI → Service (Commands)
{
 "command": "get_status" | "get_config" | "start" | "stop" | "ping",
 "request_id": "uuid",
 "args": {...}
}

Service → GUI (Responses)
{
 "success": true,
 "data": {...},
 "error": null,
 "request_id": "uuid"
}
```

**ServiceIPCCClient** in GUI connects to **IPCSERVER** in service using socket name `pyWATS_Service_{instance_id}`.

## User Workflow

1. First launch:
2. LoginWindow shown (no stored credentials)
3. Enter server URL + password/token

4. Authenticate → token encrypted and saved

5. MainWindow opens

#### 6. Subsequent launches:

7. Auto-connect using stored encrypted token

8. MainWindow opens directly if valid

9. LoginWindow only if authentication fails

#### 10. Main window:

11. Sidebar navigation (Setup, Converters, S/N Handler, etc.)

12. Page content area (QStackedWidget)

13. Status bar (connection status, instance info)

14. System tray icon (minimize to tray option)

---

## Configuration Management

---

### Configuration Hierarchy

1. Environment Variables (highest priority)

PYWATS\_SERVER\_URL

PYWATS\_API\_TOKEN

PYWATS\_INSTANCE\_ID

2. Command-line Arguments

--server-url

--instance-id

--config-file

3. Configuration File (config.json)

All settings stored in JSON

4. Defaults (hardcoded in dataclasses)

## Dynamic Configuration

### Runtime credential override:

```
In ClientConfig
def get_runtime_credentials(self):
 """Check environment variables for debug/override"""
 url = os.getenv('PYWATS_SERVER_URL') or self.service_address
 token = os.getenv('PYWATS_API_TOKEN') or self.api_token
 return url, token
```

**Use case:** Development/testing without modifying config files

## Async vs Sync Usage

### Sync API ( pyWATS )

#### Best for:

- Simple scripts
- Sequential operations
- Learning/prototyping
- Single-threaded applications

#### Example:

```
from pywats import pyWATS

api = pyWATS(
 base_url="https://company.wats.com",
 token="your-token"
)

Blocking calls
product = api.product.get_product("PRODUCT-123")
reports = api.report.get_reports(limit=10)
```

### Async API ( AsyncWATS )

#### Best for:

- High-performance applications
- Concurrent operations
- Batch processing
- Server applications

#### Example:

```

import asyncio
from pywats import AsyncWATS

async def main():
 api = AsyncWATS(
 base_url="https://company.wats.com",
 token="your-token"
)

 # Concurrent requests
 products = await asyncio.gather(
 api.product.get_product_async("PROD-1"),
 api.product.get_product_async("PROD-2"),
 api.product.get_product_async("PROD-3")
)

 asyncio.run(main())

```

**Performance benefit:** 5-100x speedup for batch operations

## When to Use Each

Scenario	Recommendation
CLI tool	Sync ( pyWATS )
Test automation script	Sync ( pyWATS )
Web server integration	Async ( AsyncWATS )
Batch data migration	Async ( AsyncWATS )
Processing 1000s of reports	Async ( AsyncWATS )
Simple report submission	Sync ( pyWATS )

## Extension Points

### 1. Custom Converters

Create custom file format converter:

```

from pywats_client.converters import FileConverter
from pywats.tools import ReportBuilder

class MyCustomConverter(FileConverter):
 @property
 def name(self) -> str:
 return "Custom Format Converter"

 @property
 def file_patterns(self) -> List[str]:
 return [".*.custom"]

 def convert(self, source, context):
 data = self._parse_custom_format(source.read_text())
 builder = ReportBuilder(
 part_number=data["part"],
 serial_number=data["sn"]
)
 # Add steps...
 return ConverterResult.success(builder.to_dict())

```

## Register in config:

```
{
 "converters": [
 {
 "name": "My Custom Converter",
 "module_path": "converters.my_custom_converter.MyCustomConverter",
 "watch_folder": "C:\\\\TestData\\\\Custom",
 "file_patterns": [".*.custom"]
 }
]
}
```

## 2. Custom Domains

### Extend pyWATS API with custom domain:

```

from pywats.core.base_repository import BaseRepository

class CustomRepository(BaseRepository):
 def get_custom_data(self, id: str):
 response = self._http_client.get(f"/api/Custom/{id}")
 return self._error_handler.handle_response(
 response, operation="get_custom_data"
)

class CustomService:
 def __init__(self, repository: CustomRepository):
 self._repo = repository

 def get_data(self, id: str):
 return self._repo.get_custom_data(id)

Register with pyWATS
api.custom = CustomService(CustomRepository(...))

```

## 3. Custom Report Steps

**Create domain-specific step types:**

```

from pywats.tools import ReportBuilder

builder = ReportBuilder(...)

Custom step with specialized validation
builder.add_custom_step(
 step_type="SpecializedTest",
 properties={
 "custom_field_1": value1,
 "custom_field_2": value2
 }
)

```

## 4. Scheduled Tasks

**Implement periodic data sync:**

```
from pywats_client.converters import ScheduledConverter

class DataSyncConverter(ScheduledConverter):
 schedule_interval_seconds = 300 # Every 5 minutes

 def execute(self, context):
 # Fetch data from external source
 # Build report
 # Return result
 pass
```

## Deployment Modes

### 1. GUI Mode (Desktop)

```
python -m pywats_client gui
or
pywats-client
```

**Use case:** Lab stations, operator-facing stations

### 2. Headless Service Mode

```
python -m pywats_client service --daemon
```

**Use case:** Production servers, headless test stations

### 3. Docker Container

```
docker run -d \
-v /path/to/config:/app/config \
-v /path/to/data:/app/data \
ghcr.io/olreppe/pywats:latest
```

**Use case:** Cloud deployments, Kubernetes, server racks

### 4. Windows Service

```
pywats-client install-service --instance-id production
```

**Use case:** Auto-start on boot, production environments

## 5. Linux systemd

```
sudo systemctl enable pywats-client@default
sudo systemctl start pywats-client@default
```

**Use case:** Server deployments, Raspberry Pi, embedded Linux

## 6. macOS launchd

```
launchctl load ~/Library/LaunchAgents/com.pywats.client.plist
```

**Use case:** Mac-based test stations

## 7. Programmatic API

```
from pywats import pyWATS

api = pyWATS(...)
api.report.submit_report(...)
```

**Use case:** Test automation scripts, custom applications

# Dependencies

---

## Core Dependencies

### pyWATS API:

- `httpx` (async HTTP client)
- `requests` (sync HTTP client)
- `pydantic` (data validation)
- `pydantic-settings` (configuration management)

### pyWATS Client:

- All API dependencies
- `watchdog` (file system monitoring)
- `cryptography` (token encryption)
- `apscheduler` (scheduled tasks)

### pyWATS GUI:

- All client dependencies
- `PySide6` (Qt6 for Python)

## Optional Dependencies

### Performance:

- `orjson` or `msgpack` (faster serialization)
- `uvloop` (faster async event loop on Linux/Mac)

### Packaging:

- `PyInstaller` (standalone executables)
- `pyinstaller-versionfile` (Windows version info)

### Development:

- `pytest` (testing)
  - `pytest-asyncio` (async test support)
  - `pytest-cov` (coverage reporting)
- 

## Design Principles

### 1. Separation of Concerns

2. API layer: communication only
3. Service layer: business logic
4. Repository layer: data access
5. GUI layer: presentation

### 6. Offline-First

7. Queue reports when network unavailable
8. Local caching of reference data
9. Persistent storage with crash recovery

### 10. Configurability

11. Everything configurable via JSON
12. Environment variable overrides

13. Command-line arguments

### 14. Extensibility

15. Custom converters via Python modules
16. Custom domains via inheritance
17. Plugin architecture for future enhancements

### 18. Reliability

19. Automatic retry with exponential backoff
20. Rate limiting to prevent server overload
21. Error handling with detailed diagnostics
22. Health monitoring and auto-reconnection

### 23. Performance

24. Async support for concurrent operations
  25. Connection pooling
  26. Optional MessagePack for faster serialization
  27. Batch operations where possible
- 

## Further Reading

---

- **Getting Started Guide** - Installation and basic usage
  - **Client Architecture** - Detailed client design
  - **Integration Patterns** - Common scenarios and workflows
  - **Error Catalog** - Error codes and remediation
  - **Docker Deployment** - Container deployment guide
  - **Domain Guides:** Report, Product, Asset, etc.
- 

**Last Updated:** January 26, 2026

**Version:** 1.3.0 (separate service/GUI architecture)

---

*Source: docs/client-architecture.md*

## pyWATS Client Architecture

---

**Version:** 1.3.0 (Service/GUI Separation)

**Last Updated:** January 2026

**Audience:** Client developers, advanced users, troubleshooters

---

# Overview

---

The pyWATS Client is a **background service** with optional **GUI frontend** for managing test station automation. Starting with v1.3.0, the client uses a **separate service and GUI architecture** with inter-process communication (IPC), enabling true headless operation and better reliability.

## Key Features:

- **Headless operation:** Service runs independently without GUI
  - **IPC communication:** GUI communicates with service via Qt LocalSocket
  - **Multi-instance:** Run multiple isolated clients on same machine
  - **Crash resilience:** Service continues running even if GUI crashes
  - **Platform support:** Windows, Linux, macOS, Docker
- 

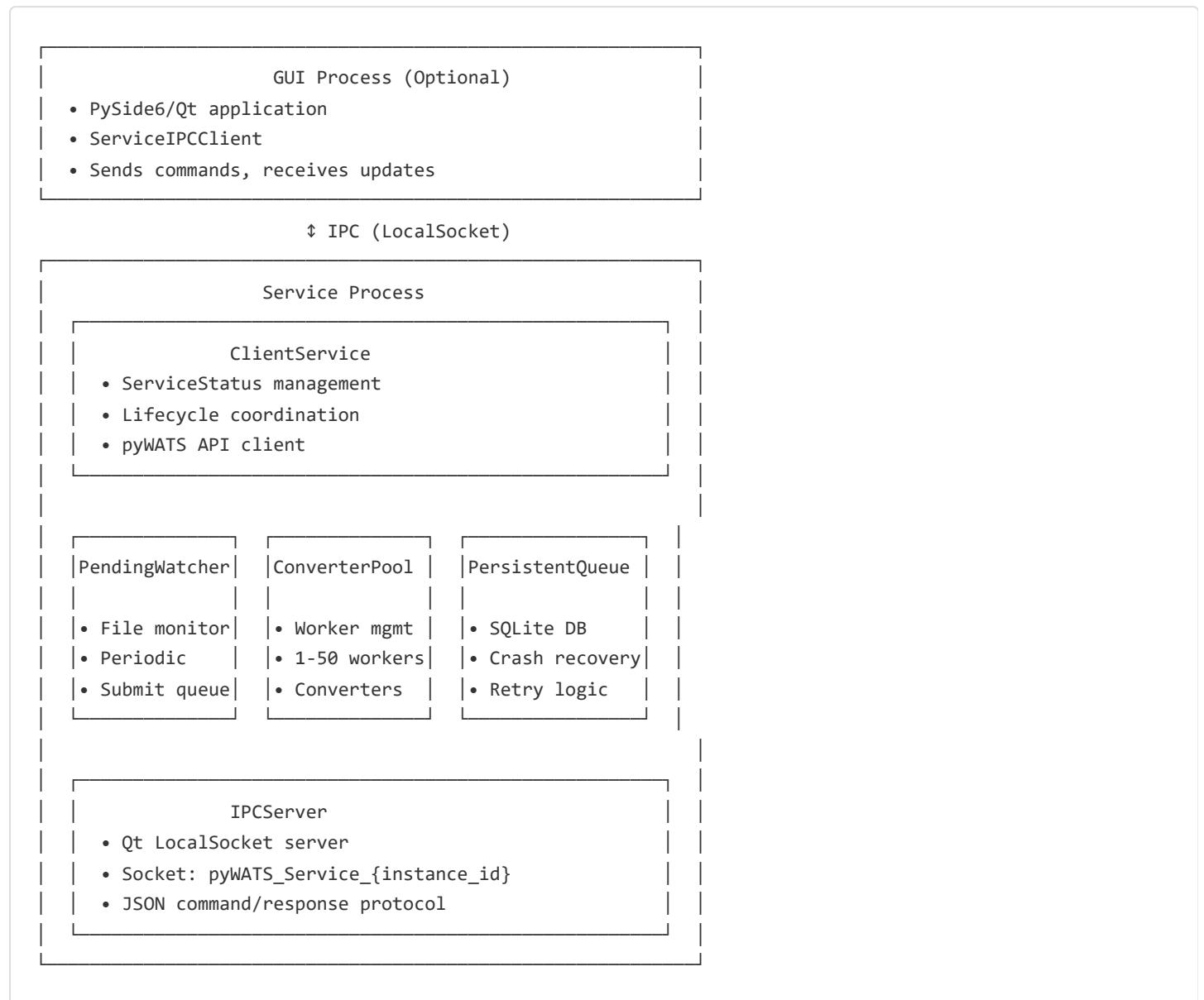
## Table of Contents

---

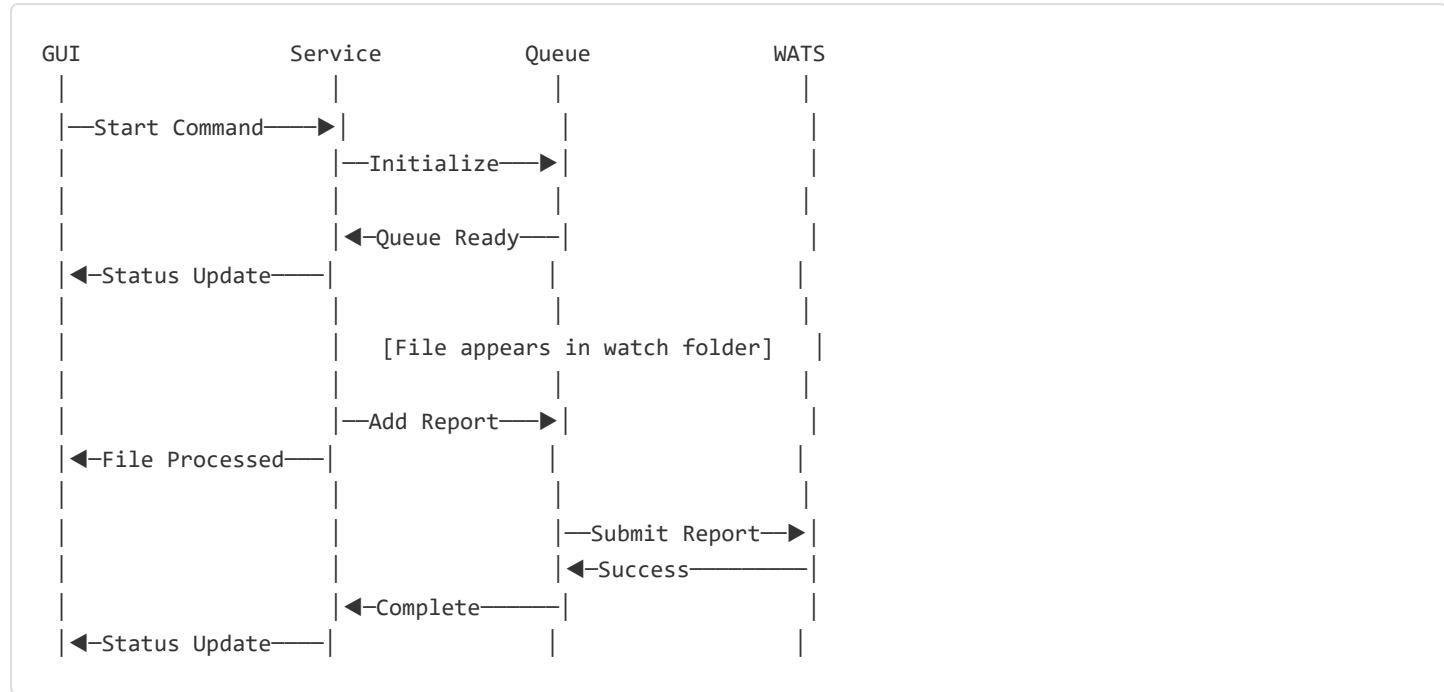
1. Architecture Overview
  2. Service Components
  3. IPC Communication
  4. Queue System
  5. Converter System
  6. File Monitoring
  7. Instance Management
  8. Configuration System
  9. Service Modes
  10. Testing Architecture
-

# Architecture Overview

## Layered Design



## Component Interactions



## Service Components

### ClientService

**Purpose:** Main service controller and lifecycle manager

**Location:** `src/pywats_client/service/client_service.py`

#### ServiceStatus States:

- STOPPED - Not running
- START\_PENDING - Initializing
- RUNNING - Active and operational
- STOP\_PENDING - Shutting down
- PAUSED - Temporarily paused (reserved for future use)

#### Key Responsibilities:

1. Initialize and coordinate all components
2. Manage service lifecycle
3. Provide pyWATS API client access
4. Track connection status
5. Handle start/stop/pause commands

#### Code Structure:

```

class ClientService:
 def __init__(self, config: ClientConfig):
 self.config = config
 self.status = ServiceStatus.STOPPED
 self._api_client: Optional[pyWATS] = None
 self._pending_watcher: Optional[PendingWatcher] = None
 self._converter_pool: Optional[ConverterPool] = None

 async def start(self):
 """Start the service and all components"""
 self.status = ServiceStatus.START_PENDING

 # Initialize API client
 self._api_client = self._create_api_client()

 # Start queue watcher
 self._pending_watcher = PendingWatcher(...)
 await self._pending_watcher.async_init()

 # Start converter pool
 self._converter_pool = ConverterPool(...)

 self.status = ServiceStatus.RUNNING

 async def stop(self):
 """Stop the service gracefully"""
 self.status = ServiceStatus.STOP_PENDING

 # Stop components in reverse order
 if self._pending_watcher:
 await self._pending_watcher.dispose()

 self.status = ServiceStatus.STOPPED

```

## API Status Tracking:

```

def get_api_status(self) -> str:
 """Get current API connection status"""
 if not self._api_client:
 return "Disconnected"

 try:
 # Test connection with quick API call
 version = self._api_client.app.get_version()
 return "Online"
 except Exception:
 return "Offline"

```

## PendingWatcher

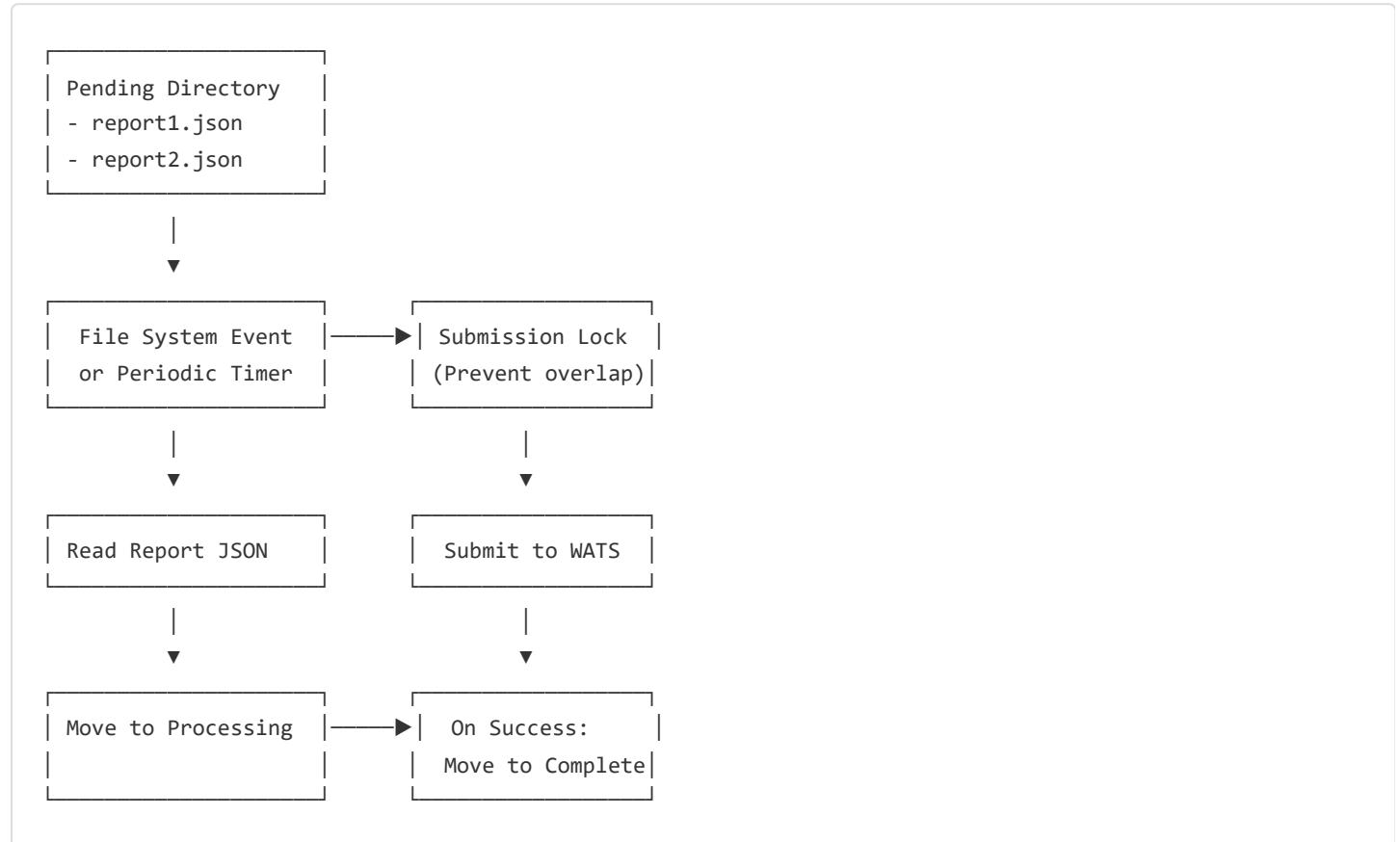
**Purpose:** Monitor and submit queued reports

**Location:** src/pywats\_client/service/pending\_watcher.py

### Key Features:

1. **File system monitoring:** Watches pending reports directory
2. **Periodic checking:** Timer-based check every 5 minutes
3. **Submission lock:** Prevents concurrent uploads
4. **Async initialization:** Non-blocking startup

### Workflow:



### Code Structure:

```

class PendingWatcher:
 def __init__(self, queue_dir: Path, api_client: pyWATS):
 self.queue_dir = queue_dir
 self.api_client = api_client
 self._timer = None
 self._submission_lock = asyncio.Lock()
 self._disposed = False

 async def async_init(self):
 """Initialize file watcher and start timer"""
 # Set up file system watcher
 self._observer = Observer()
 self._observer.schedule(
 self._event_handler,
 str(self.queue_dir / "pending"),
 recursive=False
)
 self._observer.start()

 # Start periodic check
 self._timer = asyncio.create_task(self._periodic_check())

 async def _periodic_check(self):
 """Check for pending reports every 5 minutes"""
 while not self._disposed:
 await asyncio.sleep(300) # 5 minutes
 await self._submit_pending_reports()

 async def _submit_pending_reports(self):
 """Submit all pending reports with lock"""
 async with self._submission_lock:
 # Process pending reports...
 pass

 async def dispose(self):
 """Clean shutdown"""
 self._disposed = True
 if self._observer:
 self._observer.stop()
 self._observer.join()

```

## ConverterPool

**Purpose:** Manage converter workers and execution

**Location:** src/pywats\_client/service/converter\_pool.py

### Configuration:

- max\_converter\_workers : 1-50 (default: 10)
- Worker count bounded to prevent resource exhaustion

## Responsibilities:

1. Manage worker threads/processes
2. Distribute converter tasks
3. Maintain converter registry
4. Handle pending conversion queue

## Code Structure:

```
class ConverterPool:
 def __init__(self, max_workers: int = 10):
 self.max_workers = max(1, min(50, max_workers))
 self._executor = ThreadPoolExecutor(max_workers=self.max_workers)
 self._converters: List[Converter] = []
 self._pending_queue: Queue = Queue()
 self._dispose_flag = False

 def add_converter(self, converter: Converter):
 """Register a converter"""
 self._converters.append(converter)

 def submit_conversion(self, file_path: Path, converter: Converter):
 """Submit file for conversion"""
 future = self._executor.submit(
 self._convert_file,
 file_path,
 converter
)
 return future

 def _convert_file(self, file_path: Path, converter: Converter):
 """Execute conversion (runs in worker thread)"""
 try:
 source = ConverterSource(file_path)
 result = converter.convert(source, context={})
 return result
 except Exception as e:
 return ConverterResult.failure(str(e))

 def dispose(self):
 """Shutdown pool"""
 self._dispose_flag = True
 self._executor.shutdown(wait=True)
```

## IPC Communication

### Protocol Overview

**Transport:** Qt LocalSocket (named pipe on Windows, Unix socket on Linux/Mac)

**Socket Name:** pyWATS\_Service\_{instance\_id}

- Example: pyWATS\_Service\_default
- Example: pyWATS\_Service\_production

**Message Format:** JSON

## Command Structure

**Request (GUI → Service):**

```
{
 "command": "get_status",
 "request_id": "uuid-here",
 "args": {
 "param1": "value1"
 }
}
```

**Response (Service → GUI):**

```
{
 "success": true,
 "data": {
 "status": "RUNNING",
 "api_status": "Online",
 "pending_count": 5
 },
 "error": null,
 "request_id": "uuid-here"
}
```

## Supported Commands

Command	Args	Response	Purpose
get_status	None	ServiceStatus, API status	Check service state
get_config	None	ClientConfig	Get current configuration
start	None	Success/error	Start service
stop	None	Success/error	Stop service gracefully
restart	None	Success/error	Restart service
ping	None	"pong"	Health check
get_queue_stats	None	Pending/failed counts	Queue status

## IPCSERVER Implementation

**Location:** `src/pywats_client/service/ipc_server.py`

**Code Structure:**

```

class IPCServer(QObject):
 def __init__(self, instance_id: str, service: ClientService):
 super().__init__()
 self.instance_id = instance_id
 self.service = service
 self._server = QLocalServer()
 self._clients: List[QLocalSocket] = []

 def start(self) -> bool:
 """Start IPC server"""
 socket_name = f"pyWATS_Service_{self.instance_id}"

 # Remove stale socket
 QLocalServer.removeServer(socket_name)

 # Start listening
 if not self._server.listen(socket_name):
 return False

 self._server.newConnection.connect(self._on_new_connection)
 return True

 def _on_new_connection(self):
 """Handle new client connection"""
 client = self._server.nextPendingConnection()
 client.readyRead.connect(lambda: self._on_data_ready(client))
 self._clients.append(client)

 def _on_data_ready(self, client: QLocalSocket):
 """Process incoming command"""
 data = client.readAll().data().decode('utf-8')
 request = json.loads(data)

 # Dispatch command
 response = self._handle_command(request)

 # Send response
 client.write(json.dumps(response).encode('utf-8'))
 client.flush()

 def _handle_command(self, request: dict) -> dict:
 """Execute command and return response"""
 command = request.get("command")

 if command == "get_status":
 return {
 "success": True,
 "data": {
 "status": self.service.status.name,
 "api_status": self.service.get_api_status()
 },
 "request_id": request.get("request_id")
 }

 elif command == "ping":

```

```
 return {
 "success": True,
 "data": "pong",
 "request_id": request.get("request_id")
 }

 # ... other commands
```

## ServiceIPCClient (GUI Side)

**Location:** src/pywats\_client/gui/ipc\_client.py

**Code Structure:**

```

class ServiceIPCCClient(QObject):
 status_changed = Signal(str)

 def __init__(self, instance_id: str):
 super().__init__()
 self.instance_id = instance_id
 self._socket = QLocalSocket()

 def connect_to_service(self) -> bool:
 """Connect to service IPC socket"""
 socket_name = f"pyWATS_Service_{self.instance_id}"
 self._socket.connectToServer(socket_name)

 if not self._socket.waitForConnected(5000):
 return False

 self._socket.readyRead.connect(self._on_data_ready)
 return True

 def send_command(self, command: str, args: dict = None) -> dict:
 """Send command and wait for response"""
 request = {
 "command": command,
 "request_id": str(uuid.uuid4()),
 "args": args or {}
 }

 # Send request
 data = json.dumps(request).encode('utf-8')
 self._socket.write(data)
 self._socket.flush()

 # Wait for response
 if not self._socket.waitForReadyRead(10000):
 raise TimeoutError("No response from service")

 response_data = self._socket.readAll().data().decode('utf-8')
 return json.loads(response_data)

 def get_status(self) -> dict:
 """Get service status"""
 return self.send_command("get_status")

```

## Queue System

---

### PersistentQueue Architecture

**Purpose:** SQLite-backed report queue with crash recovery

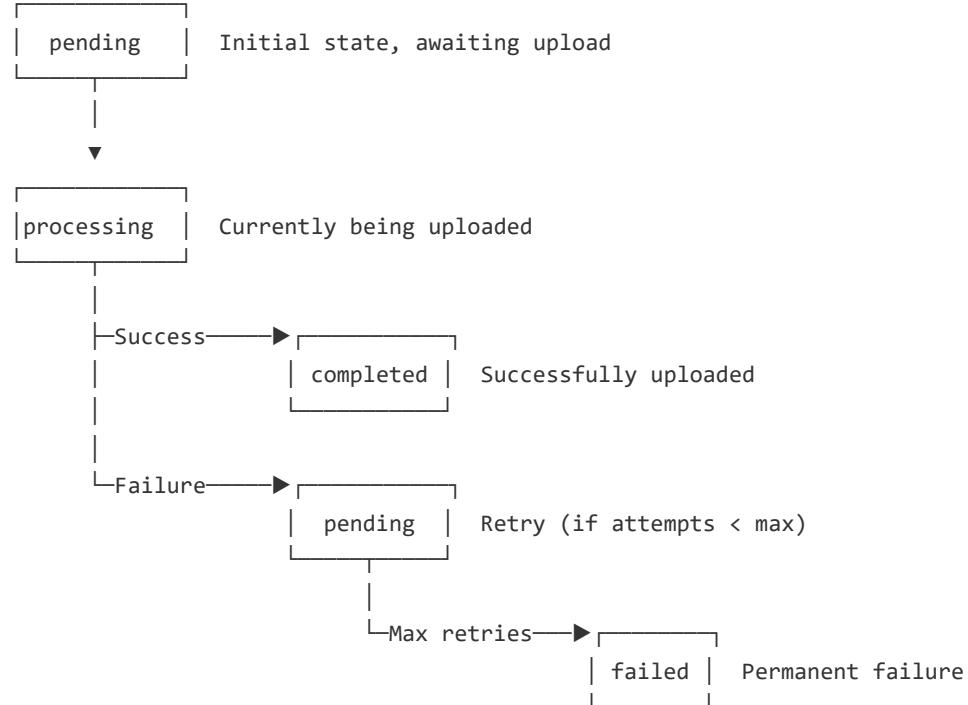
**Location:** src/pywats\_client/queue/persistent\_queue.py

## Database Schema:

```
CREATE TABLE reports (
 id TEXT PRIMARY KEY,
 report_data TEXT NOT NULL,
 status TEXT NOT NULL,
 created_at TIMESTAMP,
 updated_at TIMESTAMP,
 attempts INTEGER DEFAULT 0,
 error TEXT,
 CHECK(status IN ('pending', 'processing', 'completed', 'failed'))
);

CREATE INDEX idx_status ON reports(status);
CREATE INDEX idx_created_at ON reports(created_at);
```

## Queue States



## Crash Recovery

### On service startup:

1. Scan database for reports in processing state
2. Reset to pending state (interrupted uploads)
3. Increment attempt counter
4. Add to retry queue

### Implementation:

```

class PersistentQueue:

 def __init__(self, queue_dir: Path):
 self.queue_dir = queue_dir
 self.db_path = queue_dir / "queue.db"
 self._init_database()
 self._recover_crashed_reports()

 def _recover_crashed_reports(self):
 """Reset processing reports to pending on startup"""
 conn = sqlite3.connect(self.db_path)
 cursor = conn.cursor()

 cursor.execute("""
 UPDATE reports
 SET status = 'pending',
 attempts = attempts + 1,
 updated_at = ?
 WHERE status = 'processing'
 """, (datetime.now(),))

 recovered = cursor.rowcount
 conn.commit()
 conn.close()

 if recovered > 0:
 logger.info(f"Recovered {recovered} crashed reports")

 def add(self, report_data: dict) -> str:
 """Add report to queue"""
 report_id = str(uuid.uuid4())

 conn = sqlite3.connect(self.db_path)
 cursor = conn.cursor()

 cursor.execute("""
 INSERT INTO reports (id, report_data, status, created_at, updated_at)
 VALUES (?, ?, 'pending', ?, ?)
 """, (report_id, json.dumps(report_data), datetime.now(), datetime.now()))

 conn.commit()
 conn.close()

 return report_id

 def get_pending(self, limit: int = 10) -> List[QueuedReport]:
 """Get pending reports for upload"""
 conn = sqlite3.connect(self.db_path)
 cursor = conn.cursor()

 cursor.execute("""
 SELECT id, report_data, attempts
 FROM reports
 WHERE status = 'pending'
 ORDER BY created_at
 LIMIT ?
 """

```

```

"""", (limit,))

reports = [
 QueuedReport(
 id=row[0],
 report_data=json.loads(row[1]),
 attempts=row[2]
)
 for row in cursor.fetchall()
]

conn.close()
return reports

def mark_processing(self, report_id: str):
 """Mark report as being uploaded"""
 self._update_status(report_id, 'processing')

def mark_completed(self, report_id: str):
 """Mark report as successfully uploaded"""
 self._update_status(report_id, 'completed')

def mark_failed(self, report_id: str, error: str, max_attempts: int = 3):
 """Mark report as failed, retry if attempts < max"""
 conn = sqlite3.connect(self.db_path)
 cursor = conn.cursor()

 cursor.execute("""
 UPDATE reports
 SET status = CASE
 WHEN attempts >= ? THEN 'failed'
 ELSE 'pending'
 END,
 error = ?,
 attempts = attempts + 1,
 updated_at = ?
 WHERE id = ?
 """
 , (max_attempts, error, datetime.now(), report_id))

 conn.commit()
 conn.close()

```

## Retry Logic

### Configuration:

- max\_retry\_attempts : Default 3
- retry\_interval\_seconds : Default 60

### Exponential Backoff:

```
def calculate_retry_delay(attempt: int) -> int:
 """Calculate delay with exponential backoff"""
 base_delay = 60 # 1 minute
 max_delay = 3600 # 1 hour

 delay = min(base_delay * (2 ** attempt), max_delay)
 return delay
```

## Converter System

### Converter Lifecycle

1. Configuration Load  
↓
2. Module Import (dynamic)  
↓
3. Class Instantiation  
↓
4. File Pattern Registration  
↓
5. Watch Folder Setup  
↓  
[File appears in watch folder]  
↓
6. Event Triggered  
↓
7. Debounce Wait (500ms)  
↓
8. Pattern Match Check  
↓
9. Converter.convert() Call  
↓
10. Result Validation  
↓
11. Queue Submission or Direct Upload  
↓
12. Post-Processing (move/delete/archive)

### FileConverter Base Class

**Location:** src/pywats\_client/converters/file\_converter.py

**Abstract Methods:**

```

class FileConverter(ABC):
 @property
 @abstractmethod
 def name(self) -> str:
 """Converter display name"""
 pass

 @property
 @abstractmethod
 def version(self) -> str:
 """Converter version"""
 return "1.0.0"

 @property
 @abstractmethod
 def file_patterns(self) -> List[str]:
 """File patterns to match (glob)"""
 pass

 @abstractmethod
 def convert(self, source: ConverterSource, context: dict) -> ConverterResult:
 """Convert file to report"""
 pass

```

## Converter Configuration

```
{
 "converters": [
 {
 "name": "CSV Converter",
 "module_path": "converters.csv_converter.CSVConverter",
 "watch_folder": "C:\\\\TestData\\\\Incoming",
 "done_folder": "C:\\\\TestData\\\\Done",
 "error_folder": "C:\\\\TestData\\\\Error",
 "file_patterns": ["*.csv"],
 "post_action": "move",
 "enabled": true,
 "arguments": {
 "delimiter": ",",
 "encoding": "utf-8"
 }
 }
]
}
```

## Dynamic Loading

```
def load_converter(module_path: str, arguments: dict) -> Converter:
 """Dynamically load converter class"""
 # Parse module path
 module_name, class_name = module_path.rsplit('.', 1)

 # Import module
 module = importlib.import_module(module_name)

 # Get class
 converter_class = getattr(module, class_name)

 # Instantiate with arguments
 converter = converter_class(**arguments)

 return converter
```

## File Monitoring

### Watchdog Integration

**Library:** watchdog (cross-platform file system events)

#### Event Types:

- FileCreatedEvent - New file appears
- FileModifiedEvent - File content changed
- FileDeletedEvent - File removed
- FileMovedEvent - File renamed/moved

### Debouncing

**Problem:** File system events fire multiple times during file write

**Solution:** Wait for write completion before processing

```

class DebouncingEventHandler(FileSystemEventHandler):
 def __init__(self, callback, delay=0.5):
 self.callback = callback
 self.delay = delay # 500ms
 self._timers = {}

 def on_created(self, event):
 if event.is_directory:
 return

 # Cancel previous timer
 if event.src_path in self._timers:
 self._timers[event.src_path].cancel()

 # Start new timer
 timer = Timer(self.delay, self._process_file, [event.src_path])
 self._timers[event.src_path] = timer
 timer.start()

 def _process_file(self, file_path):
 """Called after debounce delay"""
 self.callback(file_path)
 if file_path in self._timers:
 del self._timers[file_path]

```

## Instance Management

---

### Multi-Instance Support

**Use Case:** Multiple test processes on same machine

**Example:**

- Instance "ict" - ICT testing
- Instance "fct" - Functional testing
- Instance "eol" - End-of-line testing

### Instance Isolation

**Separate per instance:**

- Configuration file: config\_{instance\_id}.json
- Queue directory: reports\_{instance\_id}/
- Log file: client\_{instance\_id}.log
- IPC socket: pyWATS\_Service\_{instance\_id}
- Lock file: instance\_{instance\_id}.lock

## Lock File Mechanism

**Location:** %TEMP%\pyWATS\_Client\instance\_{id}.lock

**Content:**

```
{
 "instance_id": "production",
 "instance_name": "Production Station",
 "pid": 12345,
 "started": "2026-01-26T10:30:00Z"
}
```

## Stale Lock Detection:

```
def _is_process_running(pid: int) -> bool:
 """Check if PID is still alive"""\n try:
 if sys.platform == "win32":
 # Windows: Check with tasklist
 output = subprocess.check_output(
 ["tasklist", "/FI", f"PID eq {pid}"],
 text=True
)
 return str(pid) in output
 else:
 # Unix: Send signal 0
 os.kill(pid, 0)
 return True
 except (subprocess.CalledProcessError, OSError):
 return False
```

## Configuration System

See Configuration Management in main architecture doc.

**Key Points:**

- JSON-based configuration
- Machine-specific encryption for API tokens
- Environment variable overrides
- Multi-instance support

# Service Modes

## 1. GUI Mode (Default)

```
python -m pywats_client gui
```

- Service + GUI in same process (legacy)
- Full user interface
- Best for: Development, troubleshooting

## 2. Service Mode (Headless)

```
python -m pywats_client service --daemon
```

- Service only, no GUI
- Runs in background
- Best for: Production, servers

## 3. Separate Service + GUI

```
Terminal 1: Start service
python -m pywats_client service

Terminal 2: Start GUI (connects to service)
python -m pywats_client gui --connect
```

- Service and GUI in separate processes
- GUI can crash without affecting service
- Best for: Reliability, debugging

## 4. Windows Service

```
pywats-client install-service --instance-id production
Start-Service pyWATS-Client-production
```

- Installed as Windows service
- Auto-start on boot
- Best for: Production Windows stations

## 5. Linux systemd

```
sudo systemctl enable pywats-client@production
sudo systemctl start pywats-client@production
```

- Managed by systemd
- Auto-restart on failure
- Best for: Production Linux stations

## 6. Docker Container

```
docker run -d ghcr.io/olreppe/pywats:client-headless
```

- Containerized deployment
- Reproducible environment
- Best for: Cloud, Kubernetes, server racks

# Testing Architecture

## Test Suite Overview

**Location:** api-tests/client/

**Coverage:** 85 tests (100% passing)

### Test Categories:

1. **Configuration (18 tests)** - Config validation, serialization, lifecycle
2. **Converters (10 tests)** - Base classes, validation, results
3. **Queue (21 tests)** - Persistence, crash recovery, retry
4. **Service (17 tests)** - ClientService, PendingWatcher, ConverterPool
5. **IPC (10 tests)** - Communication, protocols, commands
6. **Integration (9 tests)** - End-to-end workflows

## Test Philosophy

### Principles:

- Minimal mocking (use actual components where possible)
- Mock PySide6 at module level (avoid Qt runtime)
- Test business logic, not implementation details
- Focus on public interfaces

## Example Test:

```
def test_persistent_queue_crash_recovery(tmp_path):
 """Test that processing reports are recovered on restart"""
 queue = PersistentQueue(tmp_path)

 # Add report and mark as processing
 report_id = queue.add({"data": "test"})
 queue.mark_processing(report_id)

 # Simulate crash (create new queue instance)
 queue2 = PersistentQueue(tmp_path)

 # Verify report reset to pending
 pending = queue2.get_pending()
 assert len(pending) == 1
 assert pending[0].id == report_id
 assert pending[0].attempts == 1 # Incremented
```

## Running Tests

```
Run all client tests
pytest api-tests/client/ -v

Run specific category
pytest api-tests/client/test_service.py -v

Run with coverage
pytest api-tests/client/ --cov=pywats_client --cov-report=html
```

## Troubleshooting

### Service Won't Start

#### Check lock file:

```
Windows
dir "%TEMP%\pyWATS_Client*.lock"

Linux
ls /tmp/pyWATS_Client/*.lock
```

#### Remove stale lock:

```
python -m pywats_client unlock --instance-id production
```

## GUI Can't Connect to Service

### Verify service running:

```
python -m pywats_client status --instance-id production
```

### Check socket name:

```
Should be: pyWATS_Service_{instance_id}
Windows: \\.\pipe\pyWATS_Service_production
Linux: /tmp/pyWATS_Service_production
```

### Test IPC manually:

```
from pywats_client.gui.ipc_client import ServiceIPCCClient

client = ServiceIPCCClient("production")
if client.connect_to_service():
 response = client.send_command("ping")
 print(response) # Should be "pong"
```

## Queue Growing Too Large

### Check failed reports:

```
python -m pywats_client queue stats --instance-id production
```

### Retry failed:

```
python -m pywats_client queue retry --instance-id production
```

### Clear completed:

```
python -m pywats_client queue clean --instance-id production
```

## See Also

---

- **Architecture Overview** - System-wide architecture
  - **Integration Patterns** - Common workflows
  - **Client Installation** - Installation guide
  - **Service Deployment** - Service setup
- 

**Last Updated:** January 26, 2026

**Version:** 1.3.0 (Separate Service/GUI Architecture)

---

*Source: docs/integration-patterns.md*

# Integration Patterns & Common Scenarios

---

**Purpose:** Practical guidance for integrating pyWATS into manufacturing test environments

**Audience:** Integration engineers, test station setup technicians, developers

**Last Updated:** January 2026

---

## Overview

---

This guide provides **complete workflows** and **proven patterns** for common pyWATS integration scenarios. Each pattern includes configuration examples, code snippets, and troubleshooting tips.

### Quick Links:

- Complete Station Setup
  - Multi-Process Workflows
  - Error Recovery Patterns
  - Performance Optimization
  - Common Scenarios
- 

## Table of Contents

---

1. Complete Station Setup
2. Multi-Process Workflows

3. Error Recovery Patterns
  4. Performance Optimization
  5. Common Scenarios
  6. Troubleshooting Guide
- 

## Complete Station Setup

---

### Scenario: New Test Station from Scratch

**Goal:** Set up a production test station that automatically converts CSV test files and uploads to WATS.

#### Step 1: Install pyWATS Client

```
Install from PyPI
pip install pywats-api[client]

Or install from wheel
pip install pywats_api-1.3.0-py3-none-any.whl
```

#### Verify installation:

```
python -m pywats_client --version
```

#### Step 2: Initialize Configuration

##### Option A: GUI mode (recommended for first setup)

```
python -m pywats_client gui
```

- Complete login window (server URL + password)
- Configure station name and location
- Test connection

##### Option B: Headless mode

```
python -m pywats_client config init --instance-id production
```

Edit %APPDATA%\pyWATS\_Client\config\_production.json :

```
{
 "instance_id": "production",
 "instance_name": "Station ICT-01",
 "service_address": "https://company.wats.com",
 "api_token": "", // Will be encrypted after first auth
 "station_name": "ICT-STATION-01",
 "location": "Building A - Line 1",
 "purpose": "Production"
}
```

### Step 3: Authenticate and Test

**GUI mode:** Already done in login window

**Headless mode:**

```
from pywats_client.config import ClientConfig
from pywats import pyWATS

Load config
config = ClientConfig.load_for_instance("production")

Test connection
api = pyWATS(
 base_url=config.service_address,
 token=config.api_token or "YOUR_API_TOKEN_HERE"
)

Test API call
version = api.app.get_version()
print(f"Connected to WATS {version}")
```

### Step 4: Set Up Converter

**Create converter module** ( C:\Converters\csv\_converter.py ):

```

from pywats.tools import ReportBuilder
from pywats_client.converters import FileConverter, ConverterSource, ConverterResult
import csv

class CSVConverter(FileConverter):
 @property
 def name(self) -> str:
 return "CSV Test Converter"

 @property
 def file_patterns(self) -> List[str]:
 return [".csv"]

 def convert(self, source: ConverterSource, context) -> ConverterResult:
 try:
 # Parse CSV
 data = list(csv.DictReader(source.read_text().splitlines()))

 # Extract header info (first row)
 header = data[0]

 # Create report
 builder = ReportBuilder(
 part_number=header["PartNumber"],
 serial_number=header["SerialNumber"]
)

 # Add test steps
 for row in data[1:]: # Skip header row
 builder.add_step(
 name=row["TestName"],
 value=float(row["Value"]),
 unit=row.get("Unit"),
 limits=(
 float(row["LowLimit"]) if row.get("LowLimit") else None,
 float(row["HighLimit"]) if row.get("HighLimit") else None
)
)

 return ConverterResult.success(builder.to_dict())
 except Exception as e:
 return ConverterResult.failure(str(e))

```

## Add to config:

```
{
 "converters": [
 {
 "name": "CSV Test Converter",
 "module_path": "csv_converter.CSVConverter",
 "watch_folder": "C:\\\\TestData\\\\Incoming",
 "done_folder": "C:\\\\TestData\\\\Done",
 "error_folder": "C:\\\\TestData\\\\Error",
 "file_patterns": ["*.csv"],
 "post_action": "move",
 "enabled": true
 }
]
}
```

## Create directories:

```
New-Item -ItemType Directory -Force -Path "C:\\TestData\\Incoming"
New-Item -ItemType Directory -Force -Path "C:\\TestData\\Done"
New-Item -ItemType Directory -Force -Path "C:\\TestData\\Error"
```

## Step 5: Enable Offline Queue

```
{
 "offline_queue_enabled": true,
 "reports_folder": "C:\\\\TestData\\\\Queue",
 "max_retry_attempts": 5,
 "retry_interval_seconds": 60
}
```

## Queue directories created automatically:

- C:\\TestData\\Queue\\pending\\
- C:\\TestData\\Queue\\processing\\
- C:\\TestData\\Queue\\completed\\
- C:\\TestData\\Queue\\failed\\

## Step 6: Test with Sample Data

### Create test file ( C:\\TestData\\Incoming\\test.csv ):

```
PartNumber,SerialNumber,,
WIDGET-100,SN-12345,,
TestName,Value,Unit,LowLimit,HighLimit
Resistance,10.5,Ohm,9.0,12.0
Voltage,5.01,V,4.95,5.05
Current,0.998,A,0.95,1.05
```

## Monitor:

- File should be moved to Done\\ folder

- Check queue: C:\TestData\Queue\completed\
- Verify in WATS: Search for SN-12345

## Step 7: Deploy to Production

### Option A: Windows Service

```
Install as service
python -m pywats_client install-service --instance-id production

Start service
Start-Service pyWATS-Client-production

Check status
Get-Service pyWATS-Client-production
```

### Option B: Linux systemd

```
Create service file
sudo nano /etc/systemd/system/pywats-client@.service

Enable and start
sudo systemctl enable pywats-client@production
sudo systemctl start pywats-client@production

Check status
sudo systemctl status pywats-client@production
```

### Option C: Docker

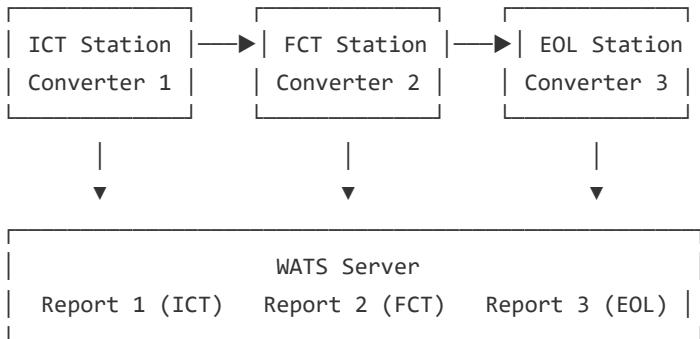
```
docker run -d \
--name pywats-client-production \
--restart unless-stopped \
-v /opt/pywats/config:/app/config \
-v /opt/pywats/data:/app/data \
-v /testdata:/testdata \
ghcr.io/olreppe/pywats:client-headless
```

## Multi-Process Workflows

### Pattern: Sequential Testing (ICT → FCT → EOL)

**Scenario:** Product goes through multiple test stages, each stage submits separate report.

## Architecture



## Implementation

**Each station runs separate client instance:**

### ICT Station:

```
{
 "instance_id": "ict",
 "instance_name": "ICT Station",
 "station_name": "ICT-STATION-01",
 "converters": [
 {"name": "ICT Converter",
 "module_path": "converters.ict_converter.ICTConverter",
 "watch_folder": "C:\\\\TestData\\\\ICT\\\\Incoming",
 "file_patterns": ["*.ict"]
 }
]
}
```

### FCT Station:

```
{
 "instance_id": "fct",
 "instance_name": "FCT Station",
 "station_name": "FCT-STATION-01",
 "converters": [
 {"name": "FCT Converter",
 "module_path": "converters.fct_converter.FCTConverter",
 "watch_folder": "C:\\\\TestData\\\\FCT\\\\Incoming",
 "file_patterns": ["*.fct"]
 }
]
}
```

### Process in converter:

```

from pywats.tools import ReportBuilder

class FCTConverter(FileConverter):
 def convert(self, source, context):
 data = self._parse_file(source.read_text())

 builder = ReportBuilder(
 part_number=data["part_number"],
 serial_number=data["serial_number"]
)

 # Set operation type (test_operation in WATS)
 builder.set_operation_type("FCT") # or "Functional Test"

 # Important: Each station creates separate report
 # WATS links them by serial_number

 return ConverterResult.success(builder.to_dict())

```

## Querying multi-process results:

```

from pywats import pyWATS

api = pyWATS(...)

Get all reports for a serial number
reports = api.report.get_reports(
 serial_number="SN-12345",
 limit=100
)

Filter by operation type
ict_reports = [r for r in reports if r.operation_type == "ICT"]
fct_reports = [r for r in reports if r.operation_type == "FCT"]
eol_reports = [r for r in reports if r.operation_type == "EOL"]

```

## Pattern: Parallel Testing (Multiple Stations, Same Process)

**Scenario:** 5 FCT stations running in parallel, all uploading to same WATS instance.

**Configuration: Same across all stations**

```
{
 "station_name": "FCT-STATION-${NUMBER}", // FCT-STATION-01, 02, 03...
 "location": "Building A - Line 1"
}
```

## Station identification:

```
import platform

Auto-detect station name from hostname
station_name = platform.node() // "FCT-STATION-03"
```

## Rate limiting consideration:

```
from pywats.core.throttle import configure_throttling

Share rate limit across all stations
500 req/min default = ~100 req/min per station with 5 stations
configure_throttling(max_requests=100, window_seconds=60)
```

# Error Recovery Patterns

## Pattern: Network Failure Handling

**Scenario:** Test station loses network connection mid-shift, queue reports, auto-upload when back online.

### Configuration

```
{
 "offline_queue_enabled": true,
 "max_retry_attempts": 5,
 "retry_interval_seconds": 60,
 "auto_reconnect": true,
 "health_check_interval": 30
}
```

### How It Works

1. **Network goes down:**
2. Client detects connection failure
3. Switches to OFFLINE status
4. Reports queue to pending/ folder

#### 5. **Network returns:**

6. Health check succeeds
7. Client reconnects automatically
8. Queued reports upload in order

9. Successful uploads move to completed/

## 10. Monitoring queue:

```
from pywats_client.queue import PersistentQueue

queue = PersistentQueue("/path/to/queue")

Check status
pending = queue.get_pending_count()
failed = queue.get_failed_count()

print(f"Pending: {pending}, Failed: {failed}")

Retry failed reports manually
failed_reports = queue.get_failed_reports()
for report in failed_reports:
 queue.retry(report.id)
```

## Pattern: Authentication Expiration

**Scenario:** API token expires or is revoked.

### Detection

```
from pywats.exceptions import AuthenticationError

try:
 api.report.submit_report(report_data)
except AuthenticationError as e:
 print(f"Authentication failed: {e}")
 # Trigger re-authentication flow
```

### Auto-renewal (GUI mode)

```
GUI detects auth failure
Shows login dialog
User re-enters credentials
Token updated and encrypted
Operations resume
```

## Manual renewal (headless mode)

```
Update token in environment
export PYWATS_API_TOKEN="new-token-here"

Or update config file
pywats-client config set --api-token "new-token"

Restart service
systemctl restart pywats-client@production
```

## Pattern: Converter Failures

**Scenario:** Converter crashes or produces invalid data.

### Error Handling in Converter

```
class RobustConverter(FileConverter):
 def convert(self, source, context):
 try:
 # Parse file
 data = self._parse_file(source.read_text())

 # Validate data
 if not self._validate(data):
 return ConverterResult.failure(
 "Validation failed: Missing required fields"
)

 # Build report
 builder = ReportBuilder(...)
 return ConverterResult.success(builder.to_dict())

 except ValueError as e:
 # Data format error
 return ConverterResult.failure(f"Parse error: {e}")

 except Exception as e:
 # Unexpected error
 context.log_error(f"Converter crash: {e}")
 return ConverterResult.failure(f"Internal error: {e}")
```

## File Routing

```
Success → done_folder/
Failure → error_folder/
```

### Review failures:

```
Check error folder
Get-ChildItem "C:\TestData\Error" -Recurse

Read error log
Get-Content "C:\TestData\Error\test.csv.error"
```

## Pattern: Crash Recovery

**Scenario:** Power failure or system crash during report upload.

### PersistentQueue Behavior

#### On startup:

1. Scan queue database (SQLite)
2. Find reports in processing state
3. Reset to pending state
4. Retry upload

#### Example recovery:

```
Automatic on service start
No manual intervention needed

To verify recovery:
from pywats_client.queue import PersistentQueue

queue = PersistentQueue(...)
queue.recover_crashed_reports()

Check recovery log
print(queue.get_recovery_stats())
```

## Performance Optimization

### Pattern: Batch Operations

**Scenario:** Upload 1000 reports from historical data.

## Async Batch Submission

```
import asyncio
from pywats import AsyncWATS

async def batch_upload(reports):
 api = AsyncWATS(
 base_url="https://company.wats.com",
 token="your-token"
)

 # Concurrent upload (limit concurrency)
 semaphore = asyncio.Semaphore(10) # Max 10 concurrent

 async def upload_one(report):
 async with semaphore:
 try:
 result = await api.report.submit_report_async(report)
 return ("success", result)
 except Exception as e:
 return ("error", str(e))

 # Execute all uploads
 results = await asyncio.gather(*[upload_one(r) for r in reports])

 # Analyze results
 successes = sum(1 for r in results if r[0] == "success")
 failures = sum(1 for r in results if r[0] == "error")

 print(f"Success: {successes}, Failures: {failures}")

Run
reports = load_reports_from_files()
asyncio.run(batch_upload(reports))
```

**Performance:** ~100x faster than sequential uploads

## Pattern: Connection Pooling

**Scenario:** Multiple test scripts running on same machine.

## Shared HTTP Client

```
from pywats import pyWATS
import atexit

Global client instance (reuses connections)
_api_client = None

def get_api():
 global _api_client
 if _api_client is None:
 _api_client = pyWATS(
 base_url="https://company.wats.com",
 token="your-token"
)
 # Clean up on exit
 atexit.register(_api_client.close)
 return _api_client

Usage in multiple scripts
api = get_api()
api.report.submit_report(report1)

Another script
api = get_api() # Reuses same connection
api.report.submit_report(report2)
```

## Pattern: Caching Reference Data

**Scenario:** Lookups for product info, process names performed frequently.

### Process Sync Cache

```
from pywats_client.service.process_sync import ProcessSyncService

Automatic caching (runs in client service)
sync = ProcessSyncService(api_client)
sync.start() # Syncs every 5 minutes

Access cached data (no network call)
processes = sync.get_processes()
levels = sync.get_levels()
product_groups = sync.get_product_groups()
```

## Manual Caching

```
import json
from functools import lru_cache

@lru_cache(maxsize=100)
def get_product_cached(api, part_number):
 return api.product.get_product(part_number)

First call: network request
product1 = get_product_cached(api, "WIDGET-100")

Second call: cached (instant)
product2 = get_product_cached(api, "WIDGET-100")
```

## Pattern: MessagePack Serialization

**Scenario:** Large reports (1000s of steps) submitted frequently.

### Configuration

```
from pywats import pyWATS

api = pyWATS(
 base_url="https://company.wats.com",
 token="your-token",
 use_msgpack=True # 2-5x faster than JSON
)

Submit large report
builder = ReportBuilder(...)
for i in range(10000):
 builder.add_step(...)

api.report.submit_report(builder.to_dict())
```

**Performance:** 2-5x smaller payload, 2-3x faster serialization

## Common Scenarios

### Scenario: Report Submission with Retry

```
from pywats import pyWATS, RetryConfig

api = pyWATS(
 base_url="https://company.wats.com",
 token="your-token",
 retry_config=RetryConfig(
 max_attempts=5,
 base_delay=2.0
)
)

Automatic retry on transient failures
try:
 result = api.report.submit_report(report_data)
 print(f"Report submitted: {result.id}")
except Exception as e:
 print(f"Failed after retries: {e}")
```

### Scenario: Process Synchronization

```
from pywats import pyWATS

api = pyWATS(...)

Fetch all processes
processes = api.process.get_processes()

Cache locally
with open("processes_cache.json", "w") as f:
 json.dump([p.model_dump() for p in processes], f)

Use cached data offline
with open("processes_cache.json") as f:
 processes = json.load(f)
```

## Scenario: Serial Number Allocation

```
from pywats import pyWATS

api = pyWATS(...)

Allocate batch of serial numbers
sns = api.production.allocate_serial_numbers(
 product_name="WIDGET-100",
 product_revision="A",
 quantity=100,
 prefix="WGT-",
 start_index=1000
)

print(sns) # ["WGT-1000", "WGT-1001", ..., "WGT-1099"]
```

## Scenario: Attachment Handling

```
from pywats import pyWATS

api = pyWATS(...)

Submit report
result = api.report.submit_report(report_data)
report_id = result.id

Add attachment (image, log file, etc.)
with open("test_image.jpg", "rb") as f:
 api.report.post_attachment(
 uut_id=report_id,
 file_name="test_image.jpg",
 file_content=f.read(),
 description="Test setup photo"
)
```

## Scenario: Box Build Assembly

```
from pywats import pyWATS
from pywats.tools import ReportBuilder

api = pyWATS(...)

Main unit report
builder = ReportBuilder(
 part_number="BOX-ASSEMBLY",
 serial_number="BOX-SN-001"
)

Add subunits
builder.add_subunit(
 part_number="PCB-MAIN",
 serial_number="PCB-001"
)
builder.add_subunit(
 part_number="POWER-SUPPLY",
 serial_number="PSU-001"
)

Add assembly tests
builder.add_step(
 name="Power-On Test",
 status="Passed"
)

Submit
api.report.submit_report(builder.to_dict())
```

## Troubleshooting Guide

### Issue: Reports Not Uploading

**Check connection status:**

```
python -m pywats_client status
```

**Check queue:**

```
Windows
dir "%APPDATA%\pyWATS_Client\reports\pending"

Linux
ls ~/.config/pywats_client/reports/pending
```

## Check logs:

```
Windows
Get-Content "%APPDATA%\pyWATS_Client\client.log" -Tail 50

Linux
tail -f ~/.config/pywats_client/client.log
```

## Manual retry:

```
from pywats_client.queue import PersistentQueue

queue = PersistentQueue("path/to/queue")
queue.retry_all_failed()
```

## Issue: Converter Not Triggering

### Verify file patterns:

```
{
 "file_patterns": ["*.csv"] // Must match file extension
}
```

### Check watch folder:

```
Ensure folder exists and is writable
Test-Path "C:\TestData\Incoming" -PathType Container
```

### Test converter manually:

```
from converters.my_converter import MyConverter

converter = MyConverter()
result = converter.convert_file("C:\\TestData\\test.csv")
print(result.success)
print(result.data or result.error)
```

## Issue: Authentication Failures

### Verify credentials:

```
Test API access
curl -H "Authorization: Basic YOUR_BASE64_TOKEN" \
https://company.wats.com/api/App/Version
```

### Check token encryption:

```
from pywats_client.config import ClientConfig

config = ClientConfig.load()
print(f"Token present: {bool(config.api_token)}")
print(f"Server URL: {config.service_address}")
```

### Re-authenticate:

```
python -m pywats_client config reset
python -m pywats_client gui # Re-enter credentials
```

## Issue: High CPU/Memory Usage

### Check converter count:

```
{
 "max_converter_workers": 5 // Reduce if needed (default: 10)
}
```

### Disable unnecessary features:

```
{
 "process_sync_enabled": false, // If not needed
 "converters_enabled": false // Temporarily disable
}
```

### Monitor resources:

```
Windows
Get-Process | Where-Object {$_.Name -like "*python*"}

Linux
top -p $(pgrep -f pywats_client)
```

# Best Practices

## 1. Always Enable Offline Queue

```
{
 "offline_queue_enabled": true
}
```

**Why:** Prevents data loss during network outages

## 2. Use Descriptive Station Names

```
{
 "station_name": "Building-A_Line-1_ICK-01",
 "location": "Building A, Line 1"
}
```

**Why:** Easier troubleshooting and reporting

## 3. Set Appropriate Retry Limits

```
{
 "max_retry_attempts": 5, // Balance persistence vs. queue growth
 "retry_interval_seconds": 60 // Avoid hammering server
}
```

## 4. Use Converter Validation

```
def convert(self, source, context):
 data = self._parse(source)

 # Validate before building report
 if not self._validate(data):
 return ConverterResult.failure("Validation failed")

 # ... build report
```

## 5. Monitor Failed Reports

```
Set up daily check
cron: 0 9 * * * python check_failed_reports.py
```

```
check_failed_reports.py
from pywats_client.queue import PersistentQueue

queue = PersistentQueue(...)
failed = queue.get_failed_count()

if failed > 10:
 send_alert(f"Warning: {failed} failed reports in queue")
```

## 6. Use Async for Batch Operations

```
DON'T: Sequential (slow)
for report in reports:
 api.report.submit_report(report)

DO: Async batch (fast)
await asyncio.gather(*[
 api.report.submit_report_async(r) for r in reports
])
```

## See Also

- **Architecture Overview** - System design and components
- **Client Architecture** - Client service details
- **Error Catalog** - Error codes and remediation
- **Docker Deployment** - Container deployment guide
- **Getting Started** - Installation and basics
- **Domain Guides:** Report, Product, Production

**Last Updated:** January 26, 2026

**Feedback:** Report issues or suggest improvements via GitHub Issues

# Domain Modules

Source: [docs/modules/README.md](#)

## Domain Module Documentation

This directory contains comprehensive documentation for each WATS domain in the pyWATS API.

### Core Domains

- **Product** - Products, revisions, BOMs, box build templates, vendors, categories
- **Asset** - Equipment tracking, calibration, maintenance, hierarchy, logs
- **Production** - Unit lifecycle, serial numbers, assembly, verification, phases
- **Report** - Test reports (UUT/UUR), all step types, querying, attachments

### Analysis & Tracking

- **Analytics** - Yield analysis, measurements, Cpk statistics, Unit Flow visualization
- **Software** - Package management, versioning, distribution, tags, virtual folders
- **RootCause** - Issue tracking, defect management, status workflows, priorities
- **Process** - Operation types, test/repair processes, caching

### Identity & Administration

- **SCIM** - User provisioning from Azure AD, SCIM protocol support

### Documentation Structure

Each domain guide includes:

- **Quick Start** - Simple examples to get started
- **API Reference** - Complete method documentation
- **Examples** - Common usage patterns and scenarios

- **Best Practices** - Performance tips and recommendations
- **Cross-references** - Links to related domains

## See Also

---

- [..../INDEX.md](#) - Main documentation index
  - [..../architecture.md](#) - System architecture overview
  - [..../getting-started.md](#) - Installation and setup
  - [..../usage/](#) - Detailed legacy module guides with comprehensive examples
- 

*Source: docs/modules/report.md*

## Report Domain

---

The Report domain manages test reports (UUT/UUR) containing test results, measurements, and step hierarchies. Test reports are the core data structure in WATS - they capture what happened when a unit was tested, including all measurements, pass/fail results, and contextual information.

## Table of Contents

---

- Quick Start
  - Core Concepts
  - UUT Reports (Unit Under Test)
  - UUR Reports (Unit Under Repair)
  - Test Steps
  - Querying Reports
  - Report Attachments
  - Advanced Usage
  - API Reference
-

# Quick Start

## Synchronous Usage

```
from pywats import pyWATS

Initialize
api = pyWATS(
 base_url="https://your-wats-server.com",
 token="your-api-token"
)

Create a UUT (test) report
report = api.report.create_uut_report(
 operator="John Smith",
 part_number="WIDGET-001",
 revision="B",
 serial_number="W12345",
 operation_type=100, # End-of-line test
 station_name="FINAL-TEST-01",
 location="Production Floor"
)

Add test steps
root = report.get_root_sequence_call()

root.add_numeric_limit_step(
 name="Supply Voltage",
 value=5.02,
 units="V",
 low_limit=4.9,
 high_limit=5.1,
 status="Passed"
)

root.add_pass_fail_step(
 name="Communication Test",
 status="Passed"
)

Submit report
report_id = api.report.submit_report(report)
print(f"Report submitted: {report_id}")

Query reports using OData filter or helper methods
Method 1: Helper method (simplest)
headers = api.report.get_headers_by_serial("W12345")

Method 2: Using query_headers with ReportType
from pywats.domains.report import ReportType
headers = api.report.query_headers(
```

```
report_type=ReportType.UUT,
odata_filter="partNumber eq 'WIDGET-001'",
```

## Asynchronous Usage

For concurrent requests and better performance:

```
import asyncio
from pywats import AsyncWATS

async def query_reports():
 async with AsyncWATS(
 base_url="https://your-wats-server.com",
 token="your-api-token"
) as api:
 # Query multiple serials concurrently
 serials = ["SN-001", "SN-002", "SN-003"]
 results = await asyncio.gather(*[
 api.report.get_headers_by_serial(sn)
 for sn in serials
])

 for sn, headers in zip(serials, results):
 print(f"{sn}: {len(headers)} reports")

 asyncio.run(query_reports())
```

```
top=100
```

```
)
```

## Method 3: Using query\_uut\_headers with OData filter

---

```
headers = api.report.query_uut_headers(
 odata_filter="partNumber eq 'WIDGET-001'",
 top=100
)
print(f"Found {len(headers)} reports")
```

## ---

## ## Core Concepts

```
UUT Report (Unit Under Test)
```

```
A **UUT Report** documents a test session for a single unit. It contains:
```

- Unit identification (part number, serial, revision)
- Station information (where tested)
- Test results (steps, measurements, pass/fail)
- Timing information (start time, execution time)
- Custom metadata

```
Key attributes:
```

- `pn`: Part number
- `sn`: Serial number
- `rev`: Revision
- `process\_code`: Operation type (e.g., 100 = EOL test)
- `result`: Overall result ("P" = Pass, "F" = Fail)
- `root`: Root SequenceCall containing all test steps

## #### Serial Number & Part Number Validation

The `sn` (serial number) and `pn` (part number) fields are validated to prevent characters that cause issues with WATS searches and filters.

```
Recommended characters: `A-Z`, `a-z`, `0-9`, `-`, `_`, `.`
```

```
Problematic characters (blocked by default): `*`, `%`, `?`, `[`, `]`, `^`, `!`, `/`, `\\`
```

```
```python
```

```
# This will raise ReportHeaderValidation
report = UUTReport(pn="PART/001", sn="SN*001", ...) # Error!
```

```
# Bypass when intentionally needed:
```

```
from pywats import allow_problematic_characters
```

```
with allow_problematic_characters():
```

```
    report = UUTReport(pn="PART/001", sn="SN*001", ...) # OK (warning issued)
```

```
# Or use SUPPRESS: prefix (stripped from value):
```

```
report = UUTReport(pn="SUPPRESS:PART/001", sn="SN-001", ...)
```

```
# report.pn == "PART/001"
```

See [ERROR_CATALOG.md](#) for details.

UUR Report (Unit Under Repair)

A **UUR Report** documents a repair operation on a failed unit. It contains:

- Reference to the original failed UUT
- Repair actions taken
- Failure analysis
- Re-test results

Key relationship:

- repair_process_code : Repair operation type (typically 500)
- test_operation_code : Original test that failed

Test Steps

Test steps are organized hierarchically:

- **SequenceCall**: Container for other steps (like a folder)
- **NumericStep**: Measurement with limits (voltage, current, etc.)
- **BooleanStep**: Pass/Fail test
- **StringStep**: Text verification
- **ChartStep**: Graphical data (waveforms, plots)
- **GenericStep**: Actions, labels, comments

Step Hierarchy

Steps form a tree structure:

```
MainSequence (root)
├─ Power Supply Tests (SequenceCall)
|   ├─ 3.3V Rail (NumericStep)
|   ├─ 5V Rail (NumericStep)
|   └─ 12V Rail (NumericStep)
├─ Communication Tests (SequenceCall)
|   ├─ UART Test (BooleanStep)
|   └─ I2C Test (BooleanStep)
└─ Final Verification (SequenceCall)
    └─ Overall Status (BooleanStep)
```

UUT Reports (Unit Under Test)

Create UUT Report

```
# Using factory method (recommended)
report = api.report.create_uut_report(
    operator="Jane Doe",
    part_number="PCB-MAIN-001",
    revision="C",
    serial_number="PCB12345",
    operation_type=50, # FCT test
    station_name="FCT-STATION-02",
    location="Factory Floor",
    purpose="Production"
)

# Set overall result
report.result = "P" # or "F" for failed

# Get root sequence to add steps
root = report.get_root_sequence_call()
```

Add Numeric Test Steps

```
# Simple numeric limit test
root.add_numeric_limit_step(
    name="3.3V Rail",
    value=3.31,
    units="V",
    low_limit=3.2,
    high_limit=3.4,
    status="Passed"
)

# Using comparison operator explicitly
from pywats.domains.report.report_models.uut.steps.comp_operator import CompOp

root.add_numeric_step(
    name="Temperature",
    value=25.5,
    unit="°C",
    comp_op=CompOp.GELE, # Greater or Equal, Less or Equal
    low_limit=20.0,
    high_limit=30.0,
    status="P"
)

# Log value only (no limits)
root.add_numeric_step(
    name="Ambient Humidity",
    value=45.2,
    unit="%",
    comp_op=CompOp.LOG, # Log only, no comparison
    status="P"
)
```

Add Boolean Test Steps

```
# Simple pass/fail
root.add_pass_fail_step(
    name="Power On Self Test",
    status="Passed"
)

# Using explicit status codes
root.add_boolean_step(
    name="Firmware Load",
    status="P"  # P=Pass, F=Fail, T=Terminated, etc.
)

# Failed step
root.add_pass_fail_step(
    name="Network Connection",
    status="Failed"
)
```

Add String Test Steps

```
# String comparison
root.add_string_step(
    name="Firmware Version",
    value="2.1.5",
    limit="2.1.5",  # Expected value
    comp_op=CompOp.CASESENSIT,  # Case-sensitive compare
    status="P"
)

# Log string value
root.add_string_step(
    name="MAC Address",
    value="00:1A:2B:3C:4D:5E",
    comp_op=CompOp.LOG,
    status="P"
)
```

Create Sequence Hierarchy

```
# Create a sequence (test group)
power_seq = root.add_sequence_call(
    name="Power Supply Tests",
    file_name="power_tests.py",
    version="1.0.0"
)

# Add tests to the sequence
power_seq.add_numeric_limit_step(
    name="3.3V Rail", value=3.31, units="V",
    low_limit=3.2, high_limit=3.4, status="Passed"
)

power_seq.add_numeric_limit_step(
    name="5V Rail", value=5.01, units="V",
    low_limit=4.9, high_limit=5.1, status="Passed"
)

# Nested sequences
comm_seq = root.add_sequence_call(name="Communication Tests")
uart_tests = comm_seq.add_sequence_call(name="UART Tests")
uart_tests.add_pass_fail_step(name="Loopback Test", status="Passed")
```

Add Misc Info (Custom Metadata)

```
# Add custom key-value metadata
report.add_misc_info("LotNumber", "LOT-2025-W01")
report.add_misc_info("WorkOrder", "WO-12345")
report.add_misc_info("Temperature", "25°C")
report.add_misc_info("Operator_Badge", "EMP-789")

# Access misc info
misc = report.misc_info
for item in misc:
    print(f"{item.key}: {item.value}")
```

Submit Report

```
# Submit to server
report_id = api.report.submit_report(report)

if report_id:
    print(f"✓ Report submitted successfully: {report_id}")
else:
    print("✗ Failed to submit report")
```

UUR Reports (Unit Under Repair)

Create UUR from Failed UUT

```
# After a unit fails a test, create UUR for repair

# 1. Create and submit failed UUT
failed_uut = api.report.create_uut_report(
    operator="Test Op",
    part_number="WIDGET-001",
    revision="A",
    serial_number="W-FAIL-001",
    operation_type=100,
    station_name="TEST-01"
)

root = failed_uut.get_root_sequence_call()
root.add_pass_fail_step(name="Power Test", status="Failed")

uut_id = api.report.submit_report(failed_uut)

# 2. Create UUR from the failed UUT
uur = api.report.create_uur_report(
    failed_uut, # Pass the UUT object
    repair_process_code=500, # Default repair code
    operator="Repair Tech",
    station_name="REPAIR-STATION-01"
)

# 3. Add repair steps
uur_root = uur.get_root_sequence_call()

uur_root.add_generic_step(
    step_type="Action",
    name="Replaced faulty capacitor C15",
    status="P"
)

uur_root.add_pass_fail_step(
    name="Re-test Power Supply",
    status="Passed"
)

# 4. Submit UUR
uur_id = api.report.submit_report(uur)
```

Create UUR from Part/Process

```
# Create UUR when you don't have the UUT object
uur = api.report.create_uur_from_part_and_process(
    part_number="WIDGET-001",
    serial_number="W-REPAIR-002",
    revision="A",
    test_operation_code=100, # Original test operation
    repair_process_code=500, # Repair operation
    operator="Repair Specialist"
)

# Add repair documentation
uur_root = uur.get_root_sequence_call()

uur_root.add_string_step(
    name="Failure Mode",
    value="No output on 12V rail",
    comp_op=CompOp.LOG,
    status="P"
)

uur_root.add_string_step(
    name="Root Cause",
    value="Damaged voltage regulator U5",
    comp_op=CompOp.LOG,
    status="P"
)

uur_root.add_string_step(
    name="Corrective Action",
    value="Replaced U5 with known good part",
    comp_op=CompOp.LOG,
    status="P"
)

api.report.submit_report(uur)
```

Failure Logging in UUR (Repair Reports)

In UUR reports, failures are logged against a *unit hierarchy* (main unit + optional sub-units). WATS uses this hierarchy to understand **where** a failure occurred and to power repair analytics.

Sub-units and idx

- A UUR report contains `subUnits` (exposed as `UURReport.sub_units`), where each element is a `UURSubUnit`.
- Every `UURSubUnit` must have an integer `idx` that is **unique within the report**.
- `idx = 0` is reserved for the **main unit** (root). The library ensures a main unit exists.
- Additional sub-units typically use `idx = 1..N`.

- `parentIdx` links a sub-unit to its parent by referencing the parent's `idx` .
 - Most users keep a single-level hierarchy where all sub-units have `parentIdx = 0` .

Failures and fail codes

- A failure is associated with a specific unit via the unit's `idx` (conceptually "part index").
- In the WSJF-style serialization used by the client formats, failures are stored under the owning `UURSubUnit.failures` . That means the failure is *implicitly* tied to that sub-unit's `idx` .
- The **repair fail code** comes from the repair type's fail-code tree (categories + selectable leaf failcodes). In the full WRML representation, WATS stores:
 - `Failcode` : the GUID of the selected leaf failcode
 - `PartIdx` : which unit/sub-unit the failure belongs to (matches the sub-unit `idx`)
 - `Idx` : an internal failure counter (separate from the sub-unit `idx`)

Practical rule of thumb: treat `idx` as a per-report "primary key" for each sub-unit, and make sure every failure is attached to the correct sub-unit so WATS can resolve `PartIdx` correctly.

Test Steps

Numeric Steps with Comparison Operators

```
from pywats.domains.report.report_models.uut.steps.comp_operator import CompOp

# GELE - Greater or Equal, Less or Equal (standard range)
root.add_numeric_step(
    name="Voltage", value=5.0, unit="V",
    comp_op=CompOp.GELE, low_limit=4.5, high_limit=5.5,
    status="P"
)

# GT - Greater Than
root.add_numeric_step(
    name="Signal Strength", value=75, unit="dBm",
    comp_op=CompOp.GT, low_limit=70,
    status="P"
)

# LT - Less Than
root.add_numeric_step(
    name="Noise Level", value=5, unit="mV",
    comp_op=CompOp.LT, high_limit=10,
    status="P"
)

# EQ - Equal
root.add_numeric_step(
    name="Counter Value", value=100,
    comp_op=CompOp.EQ, low_limit=100,
    status="P"
)

# NE - Not Equal
root.add_numeric_step(
    name="Error Code", value=0,
    comp_op=CompOp.NE, low_limit=1, # Should NOT be 1
    status="P"
)

# LOG - Log only (no comparison)
root.add_numeric_step(
    name="Timestamp", value=1640000000,
    comp_op=CompOp.LOG,
    status="P"
)
```

Multi-Measurement Steps

```
# Create a multi-measurement container
multi_num = root.add_multiple_numeric_limit_test(
    name="Power Rail Measurements"
)

# Add multiple measurements
multi_num.add_measurement(
    name="3.3V Rail", value=3.31, unit="V",
    comp_op=CompOp.GELE, low_limit=3.2, high_limit=3.4,
    status="P"
)

multi_num.add_measurement(
    name="5V Rail", value=5.01, unit="V",
    comp_op=CompOp.GELE, low_limit=4.9, high_limit=5.1,
    status="P"
)

multi_num.add_measurement(
    name="12V Rail", value=12.05, unit="V",
    comp_op=CompOp.GELE, low_limit=11.5, high_limit=12.5,
    status="P"
)

# String multi-measurement
multi_str = root.add_multiple_string_value_test(
    name="Device Information"
)

multi_str.add_measurement(
    name="Model", value="WIDGET-X1",
    comp_op=CompOp.CASESENSIT, limit="WIDGET-X1",
    status="P"
)

multi_str.add_measurement(
    name="Serial", value="SN12345",
    comp_op=CompOp.LOG,
    status="P"
)
```

Generic Steps (Actions, Comments)

```
from pywats.domains.report.report_models.uut.steps.generic_step import FlowType

# Action step
root.add_generic_step(
    step_type=FlowType.Action,
    name="Initialize Hardware",
    status="P"
)

# Label/Comment
root.add_generic_step(
    step_type=FlowType.Label,
    name="Starting communication tests...",
    status="P"
)

# Goto (control flow)
root.add_generic_step(
    step_type=FlowType.Goto,
    name="Jump to cleanup",
    status="P"
)
```

Querying Reports

The report query API uses **OData filters** for flexible querying. Helper methods are provided for common use cases.

ReportType Enum

```
from pywats.domains.report import ReportType

# ReportType.UUT = "U" - Unit Under Test (test results)
# ReportType.UUR = "R" - Unit Under Repair (repair records)
```

Query UUT Report Headers

```
# Simple queries using helper methods
headers = api.report.get_headers_by_serial("W12345")
headers = api.report.get_headers_by_part_number("WIDGET-001")
headers = api.report.get_recent_headers(days=7)
headers = api.report.get_todays_headers()

# Using the unified query_headers method
from pywats.domains.report import ReportType

headers = api.report.query_headers(
    report_type=ReportType.UUT,
    odata_filter="serialNumber eq 'W12345'",
    top=100,
    orderby="start desc"
)

# Using query_uut_headers with OData filter
headers = api.report.query_uut_headers(
    odata_filter="partNumber eq 'WIDGET-001'",
    top=100
)

print(f"Found {len(headers)} reports")
for header in headers[:10]:
    print(f"{header.serial_number}: {header.status} ({header.start_utc})")
```

OData Filter Examples

```
# Filter by serial number
headers = api.report.query_uut_headers(
    odata_filter="serialNumber eq 'W12345'"
)

# Filter by part number
headers = api.report.query_uut_headers(
    odata_filter="partNumber eq 'WIDGET-001'"
)

# Filter by status
headers = api.report.query_uut_headers(
    odata_filter="status eq 'Failed'"
)

# Date range filter
headers = api.report.query_uut_headers(
    odata_filter="start ge 2026-01-01T00:00:00Z and start le 2026-01-31T23:59:59Z"
)

# Combined filters
headers = api.report.query_uut_headers(
    odata_filter="partNumber eq 'WIDGET-001' and status eq 'Failed'"
)

# With pagination
headers = api.report.query_uut_headers(
    odata_filter="partNumber eq 'WIDGET-001'",
    top=100,
    skip=0,
    orderby="start desc"
)
```

Query with Expanded Fields

```
# Expand sub-units
headers = api.report.query_uut_headers(
    odata_filter="serialNumber eq 'W12345'",
    expand=["subUnits"]
)

for header in headers:
    if header.sub_units:
        for sub in header.sub_units:
            print(f"  Sub-unit: {sub.serial_number}")

# Expand misc info
headers = api.report.query_uut_headers(
    odata_filter="partNumber eq 'WIDGET-001'",
    expand=["miscInfo"]
)
```

Query UUR (Repair) Headers

```
# Query repair reports
repairs = api.report.query_uur_headers(
    odata_filter="serialNumber eq 'W12345'"
)

# Using ReportType enum
repairs = api.report.query_headers(
    report_type=ReportType.UUR,
    odata_filter="serialNumber eq 'W12345'"
)
```

Report Attachments

Add Attachment to Report

```
# Create report
report = api.report.create_uut_report(
    operator="Tester",
    part_number="WIDGET-001",
    revision="A",
    serial_number="W12345",
    operation_type=100
)

# Add steps...
root = report.get_root_sequence_call()
root.add_pass_fail_step(name="Test", status="Passed")

# Add attachment (e.g., screenshot, log file)
with open("test_screenshot.png", "rb") as f:
    file_data = f.read()

from pywats.domains.report.report_models import Attachment

attachment = Attachment(
    file_name="test_screenshot.png",
    mime_type="image/png",
    data=file_data
)

report.add_attachment(attachment)

# Submit with attachment
api.report.submit_report(report)
```

Download Attachments

```
# Get report header using OData filter
headers = api.report.query_uut_headers(
    odata_filter="serialNumber eq 'W12345'"
)

if headers:
    header = headers[0]

    # Get attachments for this report
    attachments = api.report.get_attachments(str(header.uuid))

    for att in attachments:
        print(f"Attachment: {att.name} ({att.size} bytes)")

        # Download attachment data
        data = api.report.download_attachment(
            str(header.uuid),
            att.name
        )

        # Save to file
        with open(f"downloaded_{att.name}", "wb") as f:
            f.write(data)
```

Advanced Usage

Complete Test Workflow

```
def run_test_and_report(serial_number, part_number, revision):
    """Complete test workflow with reporting"""

    # 1. Create report
    report = api.report.create_uut_report(
        operator="AutoTest",
        part_number=part_number,
        revision=revision,
        serial_number=serial_number,
        operation_type=100,
        station_name="AUTO-TEST-01"
    )

    root = report.get_root_sequence_call()
    all_passed = True

    # 2. Run power tests
    power_seq = root.add_sequence_call(name="Power Supply Tests")

    voltage_3v3 = measure_voltage("3V3") # Your measurement function
    power_seq.add_numeric_limit_step(
        name="3.3V Rail",
        value=voltage_3v3,
        units="V",
        low_limit=3.2,
        high_limit=3.4,
        status="Passed" if 3.2 <= voltage_3v3 <= 3.4 else "Failed"
    )

    if not (3.2 <= voltage_3v3 <= 3.4):
        all_passed = False

    # 3. Run communication tests
    comm_seq = root.add_sequence_call(name="Communication Tests")

    uart_ok = test_uart() # Your test function
    comm_seq.add_pass_fail_step(
        name="UART Loopback",
        status="Passed" if uart_ok else "Failed"
    )

    if not uart_ok:
        all_passed = False

    # 4. Set overall result
    report.result = "P" if all_passed else "F"

    # 5. Submit report
    report_id = api.report.submit_report(report)
```

```
# 6. Update production unit status
if all_passed:
    api.production.set_unit_phase(
        serial_number, part_number,
        phase="Passed"
    )
else:
    api.production.set_unit_phase(
        serial_number, part_number,
        phase="Failed"
    )

return all_passed, report_id

# Run it
passed, report_id = run_test_and_report("W12345", "WIDGET-001", "A")
print(f"Test {'PASSED' if passed else 'FAILED'} - Report: {report_id}")
```

Failure Analysis Report

```
def generate_failure_report(days=7):
    """Generate failure analysis from recent tests"""
    from datetime import datetime, timedelta

    # Get failed reports using OData filter
    start_date = datetime.now() - timedelta(days=days)
    headers = api.report.query_uut_headers(
        odata_filter=f"result eq 'Failed' and start ge {start_date.strftime('%Y-%m-%d')}",
        top=500
    )

    print(f"\n== FAILURE ANALYSIS ({days} days) ==\n")
    print(f"Total failures: {len(headers)}")

    # Group by part number
    failures_by_part = {}
    for header in headers:
        pn = header.part_number
        if pn not in failures_by_part:
            failures_by_part[pn] = []
        failures_by_part[pn].append(header)

    # Print summary
    for pn, fails in sorted(failures_by_part.items(), key=lambda x: len(x[1]), reverse=True):
        print(f"\n{pn}: {len(fails)} failures")

    # Show first few
    for header in fails[:5]:
        print(f"  {header.serial_number} - {header.start_time}")

    print("\n" + "="*50 + "\n")

# Run it
generate_failure_report(days=7)
```

API Reference

ReportService Methods

UUT Report Operations

- `create_uut_report(...)` → `UUTReport` - Create new UUT report
- `query_uut_headers(odata_filter, top, skip, orderby, expand)` → `List[ReportHeader]` - Query UUT report headers with OData
- `get_uut_report(report_id)` → `Optional[UUTReport]` - Get full report

- `submit_report(report) → Optional[UUID]` - Submit report to server

UUR Report Operations

- `create_uur_report(uut, ...) → UURReport` - Create UUR from UUT
- `create_uur_from_part_and_process(...) → UURReport` - Create UUR from metadata
- `query_uur_headers(odata_filter, top, skip, orderby, expand) → List[ReportHeader]` - Query UUR report headers with OData

Unified Query Operations

- `query_headers(report_type, odata_filter, top, skip, orderby, expand) → List[ReportHeader]` - Query UUT/UUR headers

Query Helper Methods

- `get_headers_by_serial(serial_number) → List[ReportHeader]` - Get headers by serial
- `get_headers_by_part_number(part_number) → List[ReportHeader]` - Get headers by part number
- `get_headers_by_date_range(start_date, end_date) → List[ReportHeader]` - Get headers by date range
- `get_recent_headers(count) → List[ReportHeader]` - Get most recent headers
- `get_todays_headers() → List[ReportHeader]` - Get today's headers

Attachment Operations

- `get_attachments(report_id) → List[Attachment]` - Get attachment list
- `download_attachment(report_id, filename) → bytes` - Download attachment data

Models

ReportType Enum

- `ReportType.UUT` - "U" - UUT (Unit Under Test) reports
- `ReportType.UUR` - "R" - UUR (Unit Under Repair) reports

UUTReport

- `pn` : Part number
- `sn` : Serial number
- `rev` : Revision
- `process_code` : Operation type
- `result` : "P" or "F"
- `station_name` : Station name

- root : SequenceCall (test steps)
- info : UUTInfo (metadata)

SequenceCall

- name : Sequence name
- status : Overall status
- steps : List of child steps
- Methods: add_numeric_step() , add_pass_fail_step() , add_sequence_call() , etc.

WATSFilter (Analytics API only)

Note: WATSFilter is used with the Analytics API, not for report queries.

For querying report headers, use OData filter syntax or the helper methods.

- part_number : Filter by part
- serial_number : Filter by serial
- result : "Passed" or "Failed"
- days : Last N days
- start_date_time , end_date_time : Date range
- operation_type : Operation code
- top_count : Limit results

Best Practices

1. **Always add at least one step** - Reports must have test steps
 2. **Set overall result** - Set report.result to "P" or "F"
 3. **Use meaningful step names** - Clear, descriptive names
 4. **Organize with sequences** - Group related tests
 5. **Add misc info for context** - Lot numbers, work orders, etc.
 6. **Submit promptly** - Don't delay report submission
 7. **Include units** - Always specify measurement units
 8. **Set proper limits** - Define pass/fail criteria
 9. **Log important values** - Use CompOp.LOG for reference data
 10. **Link to production** - Update unit status after testing
-

See Also

- Production Domain - Managing units and updating status
 - Analytics Domain - Analyzing test results and yield
 - Process Domain - Defining operation types
-

Source: docs/modules/asset.md

Asset Domain

The Asset domain manages test equipment, stations, fixtures, instruments, and tools used in manufacturing operations. It provides comprehensive tracking for calibration schedules, maintenance history, usage counts, hierarchical relationships (stations containing instruments), and alarm states. Assets are the physical resources that execute tests and assemblies defined in other domains.

Table of Contents

- Quick Start
 - Core Concepts
 - Asset Operations
 - Asset Types
 - Asset Hierarchy
 - Status and State Management
 - Count Tracking
 - Calibration Management
 - Maintenance Management
 - Asset Logs
 - File Operations
 - Advanced Usage
 - API Reference
-

Quick Start

Synchronous Usage

```
from pywats import pyWATS

# Initialize the API
api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your-api-token"
)

# List all assets
assets = api.asset.get_assets()
for asset in assets:
    print(f"{asset.asset_name} ({asset.serial_number})")

# Get specific asset by serial number
dmm = api.asset.get_asset_by_serial("DMM-001")
print(f"Asset: {dmm.asset_name}")
print(f"State: {dmm.state}")
print(f"Last Calibration: {dmm.last_calibration_date}")

# Create a new test instrument
from pywats.domains.asset import AssetState

# First, get an asset type
asset_types = api.asset.get_asset_types()
instrument_type = next(t for t in asset_types if "Instrument" in t.type_name)

new_asset = api.asset.create_asset(
    serial_number="SCOPE-12345",
    type_id=instrument_type.type_id,
    asset_name="Oscilloscope Rigol DS1054Z",
    description="4-channel 50MHz oscilloscope",
    location="Lab A",
    state=AssetState.OK
)
```

Asynchronous Usage

For concurrent requests and better performance:

```
import asyncio
from pywats import AsyncWATS

async def manage_assets():
    async with AsyncWATS(
        base_url="https://your-wats-server.com",
        token="your-api-token"
    ) as api:
        # Fetch asset types and assets concurrently
        asset_types, assets = await asyncio.gather(
            api.asset.get_asset_types(),
            api.asset.get_assets(top=100)
        )

        print(f"Types: {len(asset_types)}, Assets: {len(assets)}")

asyncio.run(manage_assets())
```

Record calibration

```
api.asset.record_calibration(  
    serial_number="SCOPE-12345",  
    comment="Annual calibration completed"  
)
```

Core Concepts

Asset

An **Asset** represents a piece of equipment, station, fixture, or tool used in manufacturing. Each asset is identified by a unique serial number.

Key attributes:

- `serial_number`: Unique identifier (e.g., "DMM-001", "ICT-STATION-05")
- `asset_name`: Human-readable name
- `type_id`: Links to AssetType (defines category and limits)
- `state`: Current condition (OK, WARNING, ALARM, ERROR)
- `location`: Physical location
- `parent_asset_id`: For hierarchical structures (e.g., instrument belongs to station)

Asset Type

An **AssetType** defines a category of assets with shared characteristics and limits. It specifies calibration intervals, maintenance schedules, and usage limits.

Key attributes:

- `type_name`: Category name (e.g., "Station", "Instrument", "Fixture")
- `calibration_interval`: Days between calibrations
- `maintenance_interval`: Days between maintenance
- `running_count_limit`: Max uses before maintenance
- `total_count_limit`: Lifetime usage limit

Common asset types:

- **Station**: Test station or workstation (ICT, FVT, Assembly)
- **Instrument**: Measurement equipment (DMM, Oscilloscope, Power Supply)
- **Fixture**: Test fixtures and adapters (bed-of-nails, custom jigs)
- **Software**: Software tools and licenses (LabVIEW, TestStand)
- **Tool**: Hand tools and equipment (soldering iron, torque wrench)

Asset State

Assets can be in different states based on calibration, maintenance, and usage:

- **OK**: Operating normally, within all limits
- **WARNING**: Approaching limits (calibration due soon, count near limit)
- **ALARM**: Exceeded limits (calibration overdue, count limit reached)
- **ERROR**: Hardware fault or communication failure

Asset Hierarchy

Assets can have parent-child relationships:

- A **Station** can contain multiple **Instruments** and **Fixtures**
- This models physical reality (e.g., ICT-01 station uses DMM-123 and PSU-456)
- Children increment counts when parents are used
- Helps track which equipment was used for each test

Asset Operations

List All Assets

```

```python
Get all assets
assets = api.asset.get_assets()
print(f"Total assets: {len(assets)}")

for asset in assets:
 print(f"{asset.asset_name}")
 print(f" S/N: {asset.serial_number}")
 print(f" State: {asset.state}")
 print(f" Location: {asset.location or 'Not set'}")

Filter with OData query
active_assets = api.asset.get_assets(
 filter_str="state eq 'OK'",
 top=50
)

Get assets by location
lab_assets = api.asset.get_assets(
 filter_str="contains(location,'Lab A')"
)

```

## Get Specific Asset

```

By serial number (preferred)
asset = api.asset.get_asset_by_serial("DMM-001")

By asset ID (UUID)
asset = api.asset.get_asset("550e8400-e29b-41d4-a716-446655440000")

By identifier string (tries ID first, then serial)
asset = api.asset.get_asset("DMM-001")

if asset:
 print(f"Asset: {asset.asset_name}")
 print(f"Type: {asset.type_id}")
 print(f"State: {asset.state}")
 print(f"Total Count: {asset.total_count or 0}")
 print(f"Running Count: {asset.running_count or 0}")
else:
 print("Asset not found")

```

## Create New Asset

```
from uuid import UUID
from pywats.domains.asset import AssetState

Get asset type first
types = api.asset.get_asset_types()
dmm_type = next(t for t in types if "DMM" in t.type_name.upper())

Create simple asset
asset = api.asset.create_asset(
 serial_number="DMM-NEW-001",
 type_id=dmm_type.type_id,
 asset_name="Keithley 2000 Multimeter",
 description="6.5 digit bench DMM",
 location="Lab A"
)

Create with full details
asset = api.asset.create_asset(
 serial_number="SCOPE-001",
 type_id=dmm_type.type_id,
 asset_name="Rigol DS1054Z",
 description="4-channel 50MHz oscilloscope",
 location="Lab B",
 part_number="DS1054Z",
 revision="1.0",
 state=AssetState.OK,
 total_count=0,
 running_count=0
)

print(f"Created asset ID: {asset.asset_id}")
```

## Update Existing Asset

```
Get the asset
asset = api.asset.get_asset_by_serial("DMM-001")

Modify fields
asset.asset_name = "Updated Name"
asset.location = "Lab C"
asset.description = "Moved to Lab C for project XYZ"

Save changes
updated = api.asset.update_asset(asset)
print(f"Updated: {updated.asset_name}")
```

## Delete Asset

```
Delete by serial number
success = api.asset.delete_asset(serial_number="OLD-DMM-001")

Delete by asset ID
success = api.asset.delete_asset(asset_id="550e8400-e29b-41d4-a716-446655440000")

if success:
 print("Asset deleted")
else:
 print("Delete failed - asset may be in use")
```

## Asset Types

Asset types define categories of equipment with shared characteristics.

### List Asset Types

```
Get all asset types
types = api.asset.get_asset_types()

print("Asset Types:")
for asset_type in types:
 print(f"\n{asset_type.type_name} (ID: {asset_type.type_id})")
 print(f" Calibration interval: {asset_type.calibration_interval or 'None'} days")
 print(f" Maintenance interval: {asset_type.maintenance_interval or 'None'} days")
 print(f" Running count limit: {asset_type.running_count_limit or 'None'}")
 print(f" Total count limit: {asset_type.total_count_limit or 'None'}")
```

### Create Asset Type

```
Create a new asset type
new_type = api.asset.create_asset_type(
 type_name="Thermal Chamber",
 calibration_interval=180.0, # 180 days = 6 months
 maintenance_interval=90.0, # 90 days = quarterly
 running_count_limit=5000, # Reset after 5000 cycles
 total_count_limit=50000, # Replace after 50000 cycles
 warning_threshold=0.8, # Warning at 80% of limit
 alarm_threshold=1.0 # Alarm at 100% of limit
)

print(f"Created type: {new_type.type_name} (ID: {new_type.type_id})")
```

## Get Assets by Type

```
Get all assets of a specific type
First find the type
types = api.asset.get_asset_types()
station_type = next(t for t in types if t.type_name == "Station")

Get all assets
all_assets = api.asset.get_assets()

Filter by type
stations = [a for a in all_assets if a.type_id == station_type.type_id]

print(f"Found {len(stations)} stations:")
for station in stations:
 print(f" - {station.asset_name} ({station.serial_number})")
```

## Asset Hierarchy

Assets can have parent-child relationships to model physical configurations.

### Create Asset with Parent

```
Get asset types
types = api.asset.get_asset_types()
station_type = next(t for t in types if "Station" in t.type_name)
instrument_type = next(t for t in types if "Instrument" in t.type_name)

Create parent station
station = api.asset.create_asset(
 serial_number="ICT-STATION-01",
 type_id=station_type.type_id,
 asset_name="ICT Station 1",
 location="Production Floor"
)

Create child instrument (attached to station)
dmm = api.asset.create_asset(
 serial_number="DMM-123",
 type_id=instrument_type.type_id,
 asset_name="Keithley DMM",
 parent_serial_number="ICT-STATION-01" # Link to parent
)

print(f"Created DMM as child of {station.asset_name}")
```

## Add Child to Existing Asset

```
Add a child asset to existing parent
power_supply = api.asset.add_child_asset(
 parent_serial="ICT-STATION-01",
 child_serial="PSU-456",
 child_type_id=instrument_type.type_id,
 child_name="Agilent Power Supply",
 description="Triple output bench supply",
 part_number="E3631A"
)

print(f"Added {power_supply.asset_name} to station")
```

## Get Child Assets

```
Get all children of a parent asset
parent = api.asset.get_asset_by_serial("ICT-STATION-01")

children = api.asset.get_child_assets(parent_id=str(parent.asset_id))

print(f"Station {parent.asset_name} has {len(children)} child assets:")
for child in children:
 print(f" - {child.asset_name} ({child.serial_number})"

Get children by serial number
children = api.asset.get_child_assets(parent_serial="ICT-STATION-01")

Get children at specific hierarchy level
level_1_children = api.asset.get_child_assets(
 parent_serial="ICT-STATION-01",
 level=1 # Only direct children
)
```

## Status and State Management

Assets track their operational status and can enter warning/alarm states.

## Get Asset Status

```
Get detailed status including alarm information
status = api.asset.get_status(serial_number="DMM-001")

if status:
 print(f"Asset Status:")
 print(f" State: {status.get('state')}")
 print(f" In Warning: {status.get('in_warning', False)}")
 print(f" In Alarm: {status.get('in_alarm', False)}")
 print(f" Calibration due: {status.get('calibration_due')}")
 print(f" Maintenance due: {status.get('maintenance_due')}")
 print(f" Count percentage: {status.get('count_percentage')}%)
```

## Get Asset State

```
from pywats.domains.asset import AssetState

Get current state
state = api.asset.get_asset_state(serial_number="DMM-001")

if state == AssetState.OK:
 print("Asset is operating normally")
elif state == AssetState.WARNING:
 print("Asset has warnings - check calibration/maintenance")
elif state == AssetState.ALARM:
 print("Asset in alarm state - immediate attention required")
```

## Set Asset State

```
from pywats.domains.asset import AssetState

Manually set asset state (e.g., after repair)
success = api.asset.set_asset_state(
 serial_number="DMM-001",
 state=AssetState.OK
)

Set to maintenance mode
api.asset.set_asset_state(
 serial_number="SCOPE-001",
 state=AssetState.WARNING
)
```

## Check Warning/Alarm Conditions

```
Check if asset is in warning
if api.asset.is_in_warning(serial_number="DMM-001"):
 print("Asset has warnings")

Check if asset is in alarm
if api.asset.is_in_alarm(serial_number="DMM-001"):
 print("Asset in alarm state!")

Get all assets in warning
from pywats.domains.asset import AssetAlarmState

warned_assets = api.asset.get_assets_with_alarm_state(
 alarm_states=[AssetAlarmState.WARNING]
)

print(f"Found {len(warned_assets)} assets in warning:")
for asset in warned_assets:
 print(f" - {asset.asset_name}: {asset.serial_number}")

Get all assets in alarm
alarmed_assets = api.asset.get_assets_with_alarm_state(
 alarm_states=[AssetAlarmState.ALARM]
)
```

## Count Tracking

---

Assets track usage counts for maintenance scheduling.

## Increment Count

```
Increment usage count by 1
success = api.asset.increment_count(serial_number="DMM-001")

Increment by specific amount
success = api.asset.increment_count(
 serial_number="ICT-STATION-01",
 amount=5
)

Increment parent and all children
success = api.asset.increment_count(
 serial_number="ICT-STATION-01",
 amount=1,
 increment_children=True # Also increments DMM, PSU, etc.
)

Check updated counts
asset = api.asset.get_asset_by_serial("DMM-001")
print(f"Total count: {asset.total_count}")
print(f"Running count: {asset.running_count}")
```

## Reset Running Count

```
Reset running count after maintenance
success = api.asset.reset_running_count(
 serial_number="DMM-001",
 comment="Reset after scheduled maintenance"
)

Verify reset
asset = api.asset.get_asset_by_serial("DMM-001")
print(f"Running count after reset: {asset.running_count}") # Should be 0
print(f"Total count: {asset.total_count}") # Unchanged
```

## Calibration Management

Track calibration schedules and history.

## Record Calibration

```
from datetime import datetime

Record calibration with current timestamp
success = api.asset.record_calibration(
 serial_number="DMM-001",
 comment="Annual calibration completed by Metrology Lab"
)

Record calibration with specific date
calibration_date = datetime(2024, 6, 15, 10, 30)

success = api.asset.record_calibration(
 asset_id="550e8400-e29b-41d4-a716-446655440000",
 calibration_date=calibration_date,
 comment="Calibrated against NIST traceable standards"
)

Check updated dates
asset = api.asset.get_asset_by_serial("DMM-001")
print(f"Last calibration: {asset.last_calibration_date}")
print(f"Next calibration: {asset.next_calibration_date}")
```

## Check Calibration Due

```
from datetime import datetime, timedelta

Get assets with calibration due within 30 days
all_assets = api.asset.get_assets()
now = datetime.now()
due_soon = []

for asset in all_assets:
 if asset.next_calibration_date:
 days_until_due = (asset.next_calibration_date - now).days
 if 0 <= days_until_due <= 30:
 due_soon.append((asset, days_until_due))

print(f"Assets needing calibration in next 30 days:")
for asset, days in sorted(due_soon, key=lambda x: x[1]):
 print(f" {asset.asset_name} - Due in {days} days")
```

## Calibration Reminder Report

```
def get_calibration_report(api, days_ahead=90):
 """Generate calibration schedule report"""
 from datetime import datetime, timedelta

 all_assets = api.asset.get_assets()
 now = datetime.now()
 cutoff = now + timedelta(days=days_ahead)

 # Categorize assets
 overdue = []
 due_soon = []
 upcoming = []

 for asset in all_assets:
 if not asset.next_calibration_date:
 continue

 cal_date = asset.next_calibration_date

 if cal_date < now:
 overdue.append(asset)
 elif cal_date <= cutoff:
 days_until = (cal_date - now).days
 if days_until <= 30:
 due_soon.append((asset, days_until))
 else:
 upcoming.append((asset, days_until))

 print(f"\n== CALIBRATION REPORT ({days_ahead} days) ==\n")

 if overdue:
 print(f"⚠️ OVERDUE ({len(overdue)} assets):")
 for asset in overdue:
 days_overdue = (now - asset.next_calibration_date).days
 print(f" {asset.asset_name}: {days_overdue} days overdue")

 if due_soon:
 print(f"\n⚡ DUE SOON ({len(due_soon)} assets):")
 for asset, days in sorted(due_soon, key=lambda x: x[1]):
 print(f" {asset.asset_name}: {days} days")

 if upcoming:
 print(f"\n📅 UPCOMING ({len(upcoming)} assets):")
 for asset, days in sorted(upcoming, key=lambda x: x[1])[:10]:
 print(f" {asset.asset_name}: {days} days")

 # Use it
 get_calibration_report(api, days_ahead=90)
```

# Maintenance Management

Similar to calibration, but for general maintenance.

## Record Maintenance

```
from datetime import datetime

Record maintenance
success = api.asset.record_maintenance(
 serial_number="ICT-STATION-01",
 comment="Replaced worn fixture pins, cleaned vacuum system"
)

Record with specific date
maintenance_date = datetime(2024, 7, 1, 14, 0)

success = api.asset.record_maintenance(
 serial_number="THERMAL-CHAMBER-01",
 maintenance_date=maintenance_date,
 comment="Quarterly preventive maintenance performed"
)

Check dates
asset = api.asset.get_asset_by_serial("ICT-STATION-01")
print(f"Last maintenance: {asset.last_maintenance_date}")
print(f"Next maintenance: {asset.next_maintenance_date}")
```

## Maintenance Schedule

```
def get_maintenance_schedule(api, days_ahead=60):
 """Generate maintenance schedule"""
 from datetime import datetime, timedelta

 all_assets = api.asset.get_assets()
 now = datetime.now()
 cutoff = now + timedelta(days=days_ahead)

 schedule = []

 for asset in all_assets:
 if not asset.next_maintenance_date:
 continue

 maint_date = asset.next_maintenance_date

 if now <= maint_date <= cutoff:
 days_until = (maint_date - now).days
 schedule.append((asset, days_until))

 print(f"\n== MAINTENANCE SCHEDULE ({days_ahead} days) ==\n")

 for asset, days in sorted(schedule, key=lambda x: x[1]):
 status = "⚠️ URGENT" if days <= 7 else "📅"
 print(f"{status} {asset.asset_name} ({asset.serial_number}): {days} days")

Use it
get_maintenance_schedule(api, days_ahead=60)
```

## Asset Logs

Asset logs record events and activities.

## Get Asset Logs

```
Get all logs
logs = api.asset.get_asset_log()

print(f"Total log entries: {len(logs)}")

for log in logs[:10]: # Show first 10
 print(f"{log.timestamp}: {log.message}")
 print(f" Asset: {log.asset_name}")
 print(f" User: {log.user or 'System'}")

Filter logs with OData
recent_logs = api.asset.get_asset_log(
 filter_str="timestamp gt 2024-06-01",
 top=50
)

Filter by asset
dmm_logs = api.asset.get_asset_log(
 filter_str="contains(asset_name, 'DMM')")
)
```

## Add Log Message

```
Add a log entry
asset = api.asset.get_asset_by_serial("DMM-001")

success = api.asset.add_log_message(
 asset_id=str(asset.asset_id),
 message="Repaired faulty input connector",
 user="john.smith"
)

if success:
 print("Log entry added")
```

## File Operations

### INTERNAL API - Subject to change

Assets can store configuration files, calibration certificates, manuals, etc.

## Upload File to Asset

```
Read file content
with open("calibration_cert.pdf", "rb") as f:
 content = f.read()

Upload to asset
asset = api.asset.get_asset_by_serial("DMM-001")

success = api.asset.upload_blob(
 asset_id=str(asset.asset_id),
 filename="Calibration_Certificate_2024-06-15.pdf",
 content=content
)

if success:
 print("File uploaded")
```

## List Asset Files

```
List all files for an asset
files = api.asset.list_blobs(asset_id=str(asset.asset_id))

print(f"Files for {asset.asset_name}:")
for file_info in files:
 print(f" - {file_info['filename']} ({file_info['size']} bytes)")
 print(f" Uploaded: {file_info['uploaded']}")
```

## Download File from Asset

```
Download a file
content = api.asset.download_blob(
 asset_id=str(asset.asset_id),
 filename="Calibration_Certificate_2024-06-15.pdf"
)

if content:
 # Save to local file
 with open("downloaded_cert.pdf", "wb") as f:
 f.write(content)
 print("File downloaded")
```

## Delete Files

```
Delete one or more files
success = api.asset.delete_blobs(
 asset_id=str(asset.asset_id),
 filenames=["old_manual.pdf", "outdated_config.json"]
)

if success:
 print("Files deleted")
```

# Advanced Usage

## Complete Station Setup

```
from pywats.domains.asset import AssetState

Get asset types
types = api.asset.get_asset_types()
station_type = next(t for t in types if "Station" in t.type_name)
instrument_type = next(t for t in types if "Instrument" in t.type_name)
fixture_type = next(t for t in types if "Fixture" in t.type_name)

1. Create station
station = api.asset.create_asset(
 serial_number="FVT-STATION-05",
 type_id=station_type.type_id,
 asset_name="Final Verification Test Station 5",
 location="Production Floor - Zone A",
 state=AssetState.OK
)

2. Add instruments
dmm = api.asset.create_asset(
 serial_number="DMM-FVT05-01",
 type_id=instrument_type.type_id,
 asset_name="Keithley 2000 DMM",
 parent_serial_number="FVT-STATION-05",
 part_number="2000",
 description="6.5 digit multimeter for voltage measurements"
)

scope = api.asset.create_asset(
 serial_number="SCOPE-FVT05-01",
 type_id=instrument_type.type_id,
 asset_name="Rigol DS1054Z Oscilloscope",
 parent_serial_number="FVT-STATION-05",
 part_number="DS1054Z",
 description="4-channel 50MHz scope for waveform analysis"
)

psu = api.asset.create_asset(
 serial_number="PSU-FVT05-01",
 type_id=instrument_type.type_id,
 asset_name="Agilent E3631A Power Supply",
 parent_serial_number="FVT-STATION-05",
 part_number="E3631A",
 description="Triple output bench supply"
)

3. Add fixture
fixture = api.asset.create_asset(
 serial_number="FIX-FVT05-WIDGET",
 type_id=fixture_type.type_id,
```

```
 asset_name="Widget Test Fixture",
 parent_serial_number="FVT-STATION-05",
 description="Custom fixture for Widget product family"
)

4. Record initial calibration for instruments
api.asset.record_calibration(
 serial_number="DMM-FVT05-01",
 comment="Initial calibration - factory certified"
)

api.asset.record_calibration(
 serial_number="SCOPE-FVT05-01",
 comment="Initial calibration - factory certified"
)

5. Add configuration file
fixture_config = {
 "product": "WIDGET-001",
 "pin_map": {"TP1": "V+", "TP2": "GND", "TP3": "SIGNAL"},
 "force_voltage": 5.0
}

import json
api.asset.upload_blob(
 asset_id=str(fixture.asset_id),
 filename="config.json",
 content=json.dumps(fixture_config, indent=2).encode()
)

print(f"Station {station.asset_name} set up with {len([dmm, scope, psu, fixture])} child assets")
```

## Asset Health Dashboard

```
def asset_health_dashboard(api):
 """Generate comprehensive asset health report"""
 from datetime import datetime, timedelta
 from collections import defaultdict

 all_assets = api.asset.get_assets()
 now = datetime.now()

 # Categorize assets
 stats = defaultdict(int)
 issues = {
 'cal_overdue': [],
 'cal_due_soon': [],
 'maint_overdue': [],
 'maint_due_soon': [],
 'count_warning': [],
 'count_alarm': [],
 'in_alarm': []
 }

 for asset in all_assets:
 # State
 stats[asset.state.value] += 1
 if asset.state.value == 'ALARM':
 issues['in_alarm'].append(asset)

 # Calibration
 if asset.next_calibration_date:
 days_until = (asset.next_calibration_date - now).days
 if days_until < 0:
 issues['cal_overdue'].append(asset)
 elif days_until <= 30:
 issues['cal_due_soon'].append(asset)

 # Maintenance
 if asset.next_maintenance_date:
 days_until = (asset.next_maintenance_date - now).days
 if days_until < 0:
 issues['maint_overdue'].append(asset)
 elif days_until <= 30:
 issues['maint_due_soon'].append(asset)

 # Count limits (if asset has type with limits)
 # This would require checking AssetType limits

 print("=" * 60)
 print("ASSET HEALTH DASHBOARD")
 print("=" * 60)

 print(f"\nTotal Assets: {len(all_assets)}")
 print(f"\nState Distribution:")
 for state, count in stats.items():
 print(f" {state}: {count}")
```

```
print(f"\n⚠ CRITICAL ISSUES:")
print(f" Assets in ALARM state: {len(issues['in_alarm'])}")
print(f" Calibration overdue: {len(issues['cal_overdue'])}")
print(f" Maintenance overdue: {len(issues['maint_overdue'])}")

print(f"\n📅 UPCOMING:")
print(f" Calibration due (30 days): {len(issues['cal_due_soon'])}")
print(f" Maintenance due (30 days): {len(issues['maint_due_soon'])}")

if issues['cal_overdue']:
 print(f"\n⚠ OVERDUE CALIBRATIONS:")
 for asset in issues['cal_overdue'][:5]:
 days = (now - asset.next_calibration_date).days
 print(f" {asset.asset_name}: {days} days overdue")

print("=" * 60)

Use it
asset_health_dashboard(api)
```

## Bulk Import Assets

```
def import_assets_from_csv(api, csv_file_path):
 """Import assets from CSV file"""
 import csv

 # Get asset types once
 types = api.asset.get_asset_types()
 type_map = {t.type_name: t.type_id for t in types}

 created = []

 with open(csv_file_path, 'r') as file:
 reader = csv.DictReader(file)
 for row in reader:
 type_name = row['type']
 if type_name not in type_map:
 print(f"Warning: Unknown type '{type_name}' for {row['serial_number']}")
 continue

 asset = api.asset.create_asset(
 serial_number=row['serial_number'],
 type_id=type_map[type_name],
 asset_name=row['name'],
 description=row.get('description', ''),
 location=row.get('location', ''),
 part_number=row.get('part_number'),
 revision=row.get('revision')
)

 if asset:
 created.append(asset)
 print(f"Created: {asset.asset_name}")

 return created

CSV format:
serial_number,name,type,location,description,part_number,revision
DMM-001,Keithley 2000,Instrument,Lab A,6.5 digit DMM,2000,1.0
SCOPE-001,Rigol DS1054Z,Instrument,Lab A,4-ch 50MHz,DS1054Z,1.0

imported = import_assets_from_csv(api, 'assets.csv')
print(f"Imported {len(imported)} assets")
```

## Usage Tracking with Automatic Counts

```
In your test sequence, increment counts
def run_test_sequence(api, station_serial, uut_serial):
 """Example test sequence that tracks asset usage"""

 # Increment station count (and all child instruments)
 api.asset.increment_count(
 serial_number=station_serial,
 amount=1,
 increment_children=True # Increments DMM, PSU, Scope, Fixture
)

 # Run tests...
 # test_voltage()
 # test_current()
 # test_waveform()

 # Log test completion
 station = api.asset.get_asset_by_serial(station_serial)
 api.asset.add_log_message(
 asset_id=str(station.asset_id),
 message=f"Tested unit {uut_serial}",
 user="automated_test_system"
)

 # Check if maintenance is due
 if api.asset.is_in_warning(serial_number=station_serial):
 print(f"⚠️ Station {station_serial} needs attention")
 status = api.asset.get_status(serial_number=station_serial)
 print(f"Details: {status}")

 # Use it
 run_test_sequence(api, "FVT-STATION-05", "WIDGET-001-SN12345")
```

# Practical Examples

## Serial Number Lookup with Error Handling

```
from pywats.exceptions import PyWATSError, NotFoundError, AuthenticationError

def safe_asset_lookup(api, serial_number: str) -> dict:
 """
 Robust asset lookup with comprehensive error handling.

 Returns a dict with asset data or error information.
 """

 result = {
 "success": False,
 "asset": None,
 "error": None,
 "error_type": None
 }

 try:
 # Try exact match first
 asset = api.asset.get_asset_by_serial(serial_number)

 if asset:
 result["success"] = True
 result["asset"] = asset
 return result

 # Try case-insensitive search via filter
 assets = api.asset.get_assets(
 filter_str=f"tolower(serialNumber) eq tolower('{serial_number}')",
 top=1
)

 if assets:
 result["success"] = True
 result["asset"] = assets[0]
 result["note"] = "Found via case-insensitive search"
 return result

 # Try partial match for fuzzy lookup
 assets = api.asset.get_assets(
 filter_str=f"contains(serialNumber, '{serial_number}')",
 top=5
)

 if assets:
 result["success"] = False
 result["error"] = f"Exact match not found. Did you mean: {[a.serial_number for a in assets]}"
 result["error_type"] = "PARTIAL_MATCH"
 result["suggestions"] = assets
 return result

 except PyWATSError as e:
 result["error"] = str(e)
 result["error_type"] = type(e).__name__
 return result
```

```
result["error"] = f"No asset found with serial number '{serial_number}'"
result["error_type"] = "NOT_FOUND"

except AuthenticationError as e:
 result["error"] = "Authentication failed - check API token"
 result["error_type"] = "AUTH_ERROR"
except PyWATSError as e:
 result["error"] = str(e)
 result["error_type"] = "API_ERROR"

return result

Usage
lookup = safe_asset_lookup(api, "DMM-001")

if lookup["success"]:
 asset = lookup["asset"]
 print(f"Found: {asset.asset_name} at {asset.location}")
elif lookup["error_type"] == "PARTIAL_MATCH":
 print(f"Suggestions: {[a.serial_number for a in lookup['suggestions']]}"")
else:
 print(f"Error: {lookup['error']}")
```

## Deep Asset Hierarchy Traversal

```
from typing import List, Dict, Any, Optional
from dataclasses import dataclass

@dataclass
class AssetNode:
 """Represents an asset in the hierarchy tree"""
 asset: Any # Asset object
 children: List['AssetNode']
 depth: int

def build_asset_tree(api, root_serial: str, max_depth: int = 5) -> Optional[AssetNode]:
 """
 Build a complete asset hierarchy tree from a root asset.

 Useful for:
 - Visualizing station configurations
 - Finding all equipment associated with a station
 - Generating equipment reports

 Args:
 api: pyWATS API instance
 root_serial: Serial number of root asset (e.g., station)
 max_depth: Maximum depth to traverse (prevent infinite loops)

 Returns:
 AssetNode tree or None if root not found
 """
 def _build_node(asset, depth: int) -> AssetNode:
 if depth >= max_depth:
 return AssetNode(asset=asset, children=[], depth=depth)

 # Get direct children
 children = api.asset.get_child_assets(parent_id=str(asset.asset_id))

 child_nodes = [
 _build_node(child, depth + 1)
 for child in children
]

 return AssetNode(asset=asset, children=child_nodes, depth=depth)

 root = api.asset.get_asset_by_serial(root_serial)
 if not root:
 return None

 return _build_node(root, depth=0)

def print_asset_tree(node: AssetNode, indent: str = "") -> None:
 """Pretty-print the asset hierarchy"""
 state_emoji = {
 "OK": "✓",
 "WARNING": "⚠",
 "ALARM": "⚡",
 }
```

```
"ERROR": "✗"
}

emoji = state_emoji.get(node.asset.state.value, "❓")
print(f"{indent}{emoji} {node.asset.serial_number} - {node.asset.asset_name}")

for child in node.children:
 print_asset_tree(child, indent + " | ")

def count_equipment(node: AssetNode) -> Dict[str, int]:
 """Count equipment by state in the hierarchy"""
 counts = {"total": 0, "OK": 0, "WARNING": 0, "ALARM": 0, "ERROR": 0}

 def _count(n: AssetNode):
 counts["total"] += 1
 counts[n.asset.state.value] = counts.get(n.asset.state.value, 0) + 1
 for child in n.children:
 _count(child)

 _count(node)
 return counts

Usage: Inspect a station and all its equipment
tree = build_asset_tree(api, "FVT-STATION-05")
if tree:
 print("Station Hierarchy:")
 print_asset_tree(tree)

 stats = count_equipment(tree)
 print(f"\nTotal equipment: {stats['total']}")
 print(f"OK: {stats['OK']}, Warning: {stats['WARNING']}, Alarm: {stats['ALARM']}")
```

## Concurrent Bulk Asset Operations

```
import asyncio
from typing import List, Tuple
from pywats import AsyncWATS

async def bulk_create_station_with_instruments(
 api: AsyncWATS,
 station_config: dict,
 instruments: List[dict]
) -> Tuple[bool, List[str]]:
 """
 Create a complete station setup with concurrent instrument creation.

 Much faster than sequential creation for stations with many instruments.
 """

 Args:
 api: AsyncWATS instance
 station_config: Dict with station parameters
 instruments: List of instrument configs

 Returns:
 Tuple of (success, list of serial numbers created)
 """
 errors = []
 created = []

 # Get asset types once
 types = await api.asset.get_asset_types()
 type_map = {t.type_name.lower(): t.type_id for t in types}

 # 1. Create station first (required as parent)
 try:
 station_type = type_map.get("station") or type_map.get("test station")
 station = await api.asset.create_asset(
 serial_number=station_config["serial_number"],
 type_id=station_type,
 asset_name=station_config["name"],
 location=station_config.get("location", ""),
 description=station_config.get("description", "")
)
 created.append(station.serial_number)
 except Exception as e:
 return False, [f"Station creation failed: {e}"]

 # 2. Create all instruments concurrently
 async def create_instrument(config: dict):
 type_id = type_map.get(config.get("type", "instrument").lower())
 return await api.asset.create_asset(
 serial_number=config["serial_number"],
 type_id=type_id,
 asset_name=config["name"],
 parent_serial_number=station.serial_number,
 part_number=config.get("part_number"),
 description=config.get("description", "")


```

```

)

Run all instrument creations in parallel
results = await asyncio.gather(
 *[create_instrument(inst) for inst in instruments],
 return_exceptions=True
)

for inst_config, result in zip(instruments, results):
 if isinstance(result, Exception):
 errors.append(f"{inst_config['serial_number']}: {result}")
 elif result:
 created.append(result.serial_number)

3. Record initial calibrations concurrently
calibration_tasks = [
 api.asset.record_calibration(
 serial_number=sn,
 comment="Initial calibration - setup complete"
)
 for sn in created[1:] # Skip station, calibrate instruments
]
await asyncio.gather(*calibration_tasks, return_exceptions=True)

return len(errors) == 0, created

Usage
async def setup_new_station():
 async with AsyncWATS(
 base_url="https://your-wats-server.com",
 token="your-api-token"
) as api:
 station = {
 "serial_number": "ICT-STATION-10",
 "name": "In-Circuit Test Station 10",
 "location": "Production Floor - Line 2",
 "description": "High-volume ICT station"
 }

 instruments = [
 {"serial_number": "DMM-ICT10-01", "name": "Keithley 2000", "type": "instrument",
 "part_number": "2000"},

 {"serial_number": "DMM-ICT10-02", "name": "Keithley 2000", "type": "instrument",
 "part_number": "2000"},

 {"serial_number": "PSU-ICT10-01", "name": "Agilent E3631A", "type": "instrument",
 "part_number": "E3631A"},

 {"serial_number": "RELAY-ICT10-01", "name": "NI PXI-2567", "type": "instrument",
 "part_number": "PXI-2567"},

 {"serial_number": "FIX-ICT10-MAIN", "name": "ICT Main Fixture", "type": "fixture"}
]

 success, created = await bulk_create_station_with_instruments(api, station, instruments)

 if success:
 print(f"✓ Created station with {len(created)} assets")
 for sn in created:

```

```

 print(f" - {sn}")
 else:
 print(f"❌ Errors occurred")
 for err in created: # In error case, created contains error messages
 print(f" - {err}")

asyncio.run(setup_new_station())

```

## API Reference

---

### AssetService Methods

#### Asset Operations

- `get_assets(filter_str, top) → List[Asset]` - Get all assets with optional filtering
- `get_asset(identifier) → Optional[Asset]` - Get asset by ID or serial number
- `get_asset_by_serial(serial_number) → Optional[Asset]` - Get asset by serial number
- `create_asset(...) → Optional[Asset]` - Create new asset
- `update_asset(asset) → Optional[Asset]` - Update existing asset
- `delete_asset(asset_id, serial_number) → bool` - Delete asset

#### Status Operations

- `get_status(asset_id, serial_number) → Optional[Dict]` - Get detailed status
- `get_asset_state(asset_id, serial_number) → Optional[AssetState]` - Get current state
- `set_asset_state(asset_id, serial_number, state) → bool` - Set state
- `is_in_alarm(asset_id, serial_number) → bool` - Check if in alarm
- `is_in_warning(asset_id, serial_number) → bool` - Check if in warning
- `get_assets_with_alarm_state(alarm_states, top) → List[Asset]` - Get assets in specific states

#### Count Operations

- `increment_count(asset_id, serial_number, amount, increment_children) → bool` - Increment usage count
- `reset_running_count(asset_id, serial_number, comment) → bool` - Reset running count

#### Calibration & Maintenance

- `record_calibration(asset_id, serial_number, comment, calibration_date) → bool` - Record calibration

- `record_maintenance(asset_id, serial_number, comment, maintenance_date) → bool` - Record maintenance

## Log Operations

- `get_asset_log(filter_str, top) → List[AssetLog]` - Get log entries
- `add_log_message(asset_id, message, user) → bool` - Add log entry

## Asset Type Operations

- `get_asset_types() → List[AssetType]` - Get all asset types
- `create_asset_type(...) → Optional[AssetType]` - Create new asset type

## Sub-Asset Operations

- `get_child_assets(parent_id, parent_serial, level) → List[Asset]` - Get child assets
- `add_child_asset(parent_serial, child_serial, child_type_id, ...) → Optional[Asset]` - Add child asset

## AssetServiceInternal Methods (⚠️ Subject to change)

### File Operations

- `upload_file(asset_id, filename, content) → bool` - Upload file to asset
- `list_files(asset_id) → List[Dict]` - List files for asset
- `download_file(asset_id, filename) → Optional[bytes]` - Download file
- `delete_files(asset_id, filenames) → bool` - Delete files

## Models

### Asset

- `asset_id : UUID` (auto-generated)
- `serial_number : str` (required, unique)
- `asset_name : Optional[str]`
- `type_id : UUID` (required, links to AssetType)
- `description : Optional[str]`
- `location : Optional[str]`
- `parent_asset_id : Optional[UUID]`
- `part_number : Optional[str]`
- `revision : Optional[str]`
- `state : AssetState`

- `first_seen_date` : `Optional[datetime]`
- `last_seen_date` : `Optional[datetime]`
- `last_calibration_date` : `Optional[datetime]`
- `next_calibration_date` : `Optional[datetime]`
- `last_maintenance_date` : `Optional[datetime]`
- `next_maintenance_date` : `Optional[datetime]`
- `total_count` : `Optional[int]`
- `running_count` : `Optional[int]`

## **AssetType**

- `type_id` : `UUID`
- `type_name` : `str`
- `calibration_interval` : `Optional[float]` (days)
- `maintenance_interval` : `Optional[float]` (days)
- `running_count_limit` : `Optional[int]`
- `total_count_limit` : `Optional[int]`
- `warning_threshold` : `Optional[float]` (0.0 - 1.0)
- `alarm_threshold` : `Optional[float]` (0.0 - 1.0)

## **AssetState (Enum)**

- `OK` : Normal operation
- `WARNING` : Approaching limits
- `ALARM` : Limits exceeded
- `ERROR` : Hardware fault

## **AssetAlarmState (Enum)**

- `OK` : No issues
- `WARNING` : Warning conditions
- `ALARM` : Alarm conditions

## **AssetLog**

- `log_id` : `UUID`
- `asset_id` : `UUID`
- `asset_name` : `str`
- `timestamp` : `datetime`
- `message` : `str`

- user : Optional[str]
- 

## Best Practices

---

1. **Use serial numbers consistently** - They're the primary identifier, make them meaningful
  2. **Define asset types first** - Create asset types before creating assets
  3. **Set calibration/maintenance intervals** - Define intervals in AssetType to enable automatic scheduling
  4. **Use hierarchies for accuracy** - Model physical relationships (station → instruments)
  5. **Increment children when using parent** - Set `increment_children=True` when incrementing station counts
  6. **Track usage in test sequences** - Automatically increment counts during tests
  7. **Reset running count after maintenance** - Clear running count to restart interval
  8. **Add meaningful log messages** - Document repairs, config changes, relocations
  9. **Monitor warning states** - Check warnings before they become alarms
  10. **Store calibration certificates** - Use file operations to attach cal certs and manuals
  11. **Regular health checks** - Run dashboard reports to catch issues early
  12. **Don't delete assets** - Set state to inactive instead, preserve history
-

# Common Workflows

---

## Monthly Calibration Planning

```
from datetime import datetime, timedelta

Get all assets needing calibration in next 60 days
all_assets = api.asset.get_assets()
now = datetime.now()
cutoff = now + timedelta(days=60)

cal_schedule = []

for asset in all_assets:
 if asset.next_calibration_date and asset.next_calibration_date <= cutoff:
 days = (asset.next_calibration_date - now).days
 cal_schedule.append((asset, days))

Sort by urgency
cal_schedule.sort(key=lambda x: x[1])

print("== CALIBRATION SCHEDULE (60 days) ==")
for asset, days in cal_schedule:
 urgency = "🔴 OVERDUE" if days < 0 else "⚠️ URGENT" if days <= 7 else "🕒"
 print(f"{urgency} {asset.asset_name}: {abs(days)} days {'overdue' if days < 0 else 'remaining'}")
```

## Post-Maintenance Workflow

```
After completing maintenance
asset_serial = "ICT-STATION-01"

1. Record maintenance
api.asset.record_maintenance(
 serial_number=asset_serial,
 comment="Quarterly PM: cleaned, lubricated, replaced worn parts"
)

2. Reset running count
api.asset.reset_running_count(
 serial_number=asset_serial,
 comment="Count reset after quarterly maintenance"
)

3. Set state to OK
from pywats.domains.asset import AssetState
api.asset.set_asset_state(
 serial_number=asset_serial,
 state=AssetState.OK
)

4. Add log entry
asset = api.asset.get_asset_by_serial(asset_serial)
api.asset.add_log_message(
 asset_id=str(asset.asset_id),
 message="Station returned to service after quarterly maintenance",
 user="maintenance.team"
)

print(f"{asset_serial} maintenance complete and returned to service")
```

## Troubleshooting

### Asset not found

```
asset = api.asset.get_asset_by_serial("DMM-001")
if not asset:
 print("Asset doesn't exist - check serial number or create it")
```

## Cannot create asset without type

```
Always get type_id first
types = api.asset.get_asset_types()
if not types:
 print("No asset types defined - create types first")
else:
 type_id = types[0].type_id
 asset = api.asset.create_asset(serial_number="...", type_id=type_id, ...)
```

## Calibration/maintenance dates not updating

```
Make sure you're calling the right method
api.asset.record_calibration(serial_number="...", comment="...")
NOT: api.asset.post_calibration() (that's lower level)

Verify update
asset = api.asset.get_asset_by_serial("...")
print(f"Last cal: {asset.last_calibration_date}")
```

## See Also

- Production Domain - Using assets in production tests
- Report Domain - Including station info in test reports
- Process Domain - Defining test operations that use assets

*Source: docs/modules/process.md*

## Process Domain

The Process domain provides access to process and operation type definitions. Processes define the types of operations performed on units (e.g., Test, Repair, Assembly). This domain uses an in-memory cache with configurable refresh intervals to optimize performance by reducing API calls. It provides read-only access to process definitions.

## Table of Contents

- Quick Start

- Core Concepts
  - Process Operations
  - Cache Management
  - Operation Types
  - Advanced Usage
  - API Reference
- 

## Quick Start

---

### Synchronous Usage

```
from pywats import pyWATS

Initialize
api = pyWATS(
 base_url="https://your-wats-server.com",
 token="your-api-token"
)

Get test operation
test_op = api.process.get_test_operation("ICT")

if test_op:
 print(f"Operation: {test_op.name}")
 print(f"Code: {test_op.code}")
 print(f"Type: {test_op.process_type}")

Get repair operation
repair_op = api.process.get_repair_operation("Rework")

Get operation by name or code
operation = api.process.get_operation("FCT") # Can be name or code

Refresh cache manually
api.process.refresh()

print(f"Cache last refreshed: {api.process.last_refresh}")
print(f"Refresh interval: {api.process.refresh_interval} seconds")
```

### Asynchronous Usage

For concurrent requests and better performance:

```

import asyncio
from pywats import AsyncWATS

async def get_processes():
 async with AsyncWATS(
 base_url="https://your-wats-server.com",
 token="your-api-token"
) as api:
 # Get multiple operations concurrently
 ict, fct, repair = await asyncio.gather(
 api.process.get_test_operation("ICT"),
 api.process.get_test_operation("FCT"),
 api.process.get_repair_operation("Rework")
)

 print(f"ICT code: {ict.code if ict else 'N/A'}")
 print(f"FCT code: {fct.code if fct else 'N/A'}")

 asyncio.run(get_processes())

```

## Core Concepts

---

### Process (Operation Type)

A **Process** defines a type of operation:

- name : Operation name (e.g., "In-Circuit Test")
- code : Short code (e.g., "ICT")
- process\_type : Type (TEST, REPAIR, ASSEMBLY, etc.)
- description : Operation description

### Process Types

Common process types:

- **TEST**: Testing operations (ICT, FCT, Functional Test)
- **REPAIR**: Repair and rework operations
- **ASSEMBLY**: Assembly and integration
- **CALIBRATION**: Calibration operations
- **INSPECTION**: Visual or automated inspection

### Caching

The Process service uses in-memory caching:

- **Default refresh**: 300 seconds (5 minutes)
- **Auto-refresh**: Cache refreshes when age exceeds interval

- **Thread-safe:** Uses locks for concurrent access
  - **Performance:** Reduces API calls for frequently accessed data
- 

## Process Operations

---

### Get Operation by Code or Name

```
Get by code
ict = api.process.get_operation("ICT")

if ict:
 print(f"Name: {ict.name}")
 print(f"Code: {ict.code}")
 print(f"Type: {ict.process_type}")
 print(f"Description: {ict.description}")

Get by name
fct = api.process.get_operation("Functional Test")

if fct:
 print(f"Found: {fct.name} ({fct.code})")
```

### Get Test Operations

```
Get test operation specifically
test_op = api.process.get_test_operation("ICT")

if test_op:
 print(f"Test Operation: {test_op.name}")
 print(f"Code: {test_op.code}")
else:
 print("Test operation not found")

Try by name or code
fct = api.process.get_test_operation("FCT")
functional = api.process.get_test_operation("Functional Test")

Both should return the same operation
if fct and functional:
 assert fct.code == functional.code
 print(f"Found: {fct.name}")
```

## Get Repair Operations

```
Get repair operation
repair = api.process.get_repair_operation("Rework")

if repair:
 print(f"Repair Operation: {repair.name}")
 print(f"Code: {repair.code}")
 print(f"Type: {repair.process_type}")

Common repair operations
repair_codes = ["Rework", "Repair", "Debug"]

for code in repair_codes:
 op = api.process.get_repair_operation(code)
 if op:
 print(f" {code}: {op.name}")
```

## List All Operations

```
Get all operations (via cache)
Note: This accesses internal cache - implementation may vary

all_operations = api.process.get_all_operations()

print(f"== ALL OPERATIONS ({len(all_operations)}) ==\n")

Group by type
by_type = {}
for op in all_operations:
 op_type = op.process_type
 if op_type not in by_type:
 by_type[op_type] = []
 by_type[op_type].append(op)

for op_type, ops in sorted(by_type.items()):
 print(f"{op_type}:")
 for op in ops:
 print(f" {op.code}: {op.name}")
 print()
```

# Cache Management

## Check Cache Status

```
from datetime import datetime

Get cache info
last_refresh = api.process.last_refresh
refresh_interval = api.process.refresh_interval

print(f"==> CACHE STATUS ==>")
print(f"Last refresh: {last_refresh}")
print(f"Refresh interval: {refresh_interval} seconds ({refresh_interval/60:.1f} minutes)")

Calculate cache age
if last_refresh:
 age = (datetime.now() - last_refresh).total_seconds()
 print(f"Cache age: {age:.0f} seconds")

 if age > refresh_interval:
 print("⚠ Cache is stale and will refresh on next access")
 else:
 remaining = refresh_interval - age
 print(f"✓ Cache is fresh ({remaining:.0f} seconds until refresh)")
else:
 print("Cache not initialized")
```

## Manual Refresh

```
Force cache refresh
print("Refreshing process cache...")
api.process.refresh()

print(f"Cache refreshed at: {api.process.last_refresh}")
```

## Configure Refresh Interval

```
Set custom refresh interval (in seconds)
10 minutes = 600 seconds
api.process.refresh_interval = 600

print(f"Refresh interval set to {api.process.refresh_interval} seconds")

For frequently changing process definitions, use shorter interval
1 minute = 60 seconds
api.process.refresh_interval = 60

For stable definitions, use longer interval
1 hour = 3600 seconds
api.process.refresh_interval = 3600
```

## Auto-Refresh Behavior

```
import time

Cache auto-refreshes when age exceeds interval
Set short interval for demo
api.process.refresh_interval = 5 # 5 seconds

First access - loads cache
op1 = api.process.get_operation("ICT")
print(f"First access: {api.process.last_refresh}")

Wait for cache to expire
time.sleep(6)

Next access - auto-refreshes
op2 = api.process.get_operation("FCT")
print(f"After expiry: {api.process.last_refresh}")

Reset to default
api.process.refresh_interval = 300
```

# Operation Types

## Using Processes in Production

```
When creating production records, reference processes

from pywats.domains.production.models import UnitInfo

Get the test operation
test_op = api.process.get_test_operation("ICT")

Use in unit creation or update
unit = api.production.get_unit("SN12345")

if unit and test_op:
 # Record that unit went through this process
 # (actual method depends on Production API)
 print(f"Unit {unit.serial_number} processed through {test_op.name}")
```

## Using Processes in Reports

```
When creating UUT reports, reference the operation

from pywats.domains.report.models import UUTReport

Get operation
operation = api.process.get_test_operation("FCT")

Create report with operation reference
report = UUTReport(
 serial_number="SN12345",
 part_number="WIDGET-001",
 operation_type_code=operation.code, # Reference operation
 station="FCT-01"
)

Submit report
api.report.submit_uut_report(report)
```

## Validation Helper

```
def validate_operation_code(code):
 """Validate that an operation code exists"""

 operation = api.process.get_operation(code)

 if operation:
 print(f"\u2713 Valid operation: {operation.name} ({operation.code})")
 return True
 else:
 print(f"\u2717 Invalid operation code: {code}")
 return False

Use it
validate_operation_code("ICT") # Valid
validate_operation_code("INVALID") # Invalid
```

## Advanced Usage

### Operation Lookup Table

```
def build_operation_lookup():
 """Build quick lookup table for operations"""

 # Get all operations
 all_ops = api.process.get_all_operations()

 # Build lookup by code
 by_code = {op.code: op for op in all_ops}

 # Build lookup by name (lowercase for case-insensitive)
 by_name = {op.name.lower(): op for op in all_ops}

 return by_code, by_name

Use it
code_lookup, name_lookup = build_operation_lookup()

Fast lookups
ict = code_lookup.get("ICT")
fct = name_lookup.get("functional test")

print(f"ICT: {ict.name if ict else 'Not found'}")
print(f"FCT: {fct.name if fct else 'Not found'}")
```

## Process Type Report

```
def process_type_report():
 """Generate report of operations by type"""

 all_ops = api.process.get_all_operations()

 # Group by type
 by_type = {}
 for op in all_ops:
 op_type = op.process_type
 if op_type not in by_type:
 by_type[op_type] = []
 by_type[op_type].append(op)

 print("=" * 70)
 print("PROCESS TYPE REPORT")
 print("=" * 70)

 for op_type in sorted(by_type.keys()):
 ops = by_type[op_type]
 print(f"\n{op_type} ({len(ops)} operations):")

 for op in sorted(ops, key=lambda x: x.code):
 print(f" {op.code:<10} {op.name}")

 print("\n" + "=" * 70)
 print(f"Total: {len(all_ops)} operations")
 print("=" * 70)

Use it
process_type_report()
```

## Find Operations by Prefix

```
def find_operations_by_prefix(prefix):
 """Find operations with codes starting with prefix"""

 all_ops = api.process.get_all_operations()

 matching = [
 op for op in all_ops
 if op.code.startswith(prefix.upper()))
]

 print(f"==== OPERATIONS STARTING WITH '{prefix}' ({len(matching)}) ====")

 for op in matching:
 print(f"{op.code}: {op.name}")
 print(f" Type: {op.process_type}")

Use it
find_operations_by_prefix("T") # All test operations
find_operations_by_prefix("R") # All repair operations
```

## Operation Usage Tracking

```
def track_operation_usage(operation_code, days=7):
 """Track how often an operation is used in reports"""
 from datetime import datetime, timedelta

 # Verify operation exists
 operation = api.process.get_operation(operation_code)

 if not operation:
 print(f"Operation '{operation_code}' not found")
 return

 # Query reports with this operation using OData
 headers = api.report.query_uut_headers(
 odata_filter=f"processCode eq {operation_code}",
 top=1000
)

 print(f"==== USAGE: {operation.name} ({operation.code}) ===")
 print(f"Period: Last {days} days")
 print(f"Reports: {len(headers)}")

 # Breakdown by station
 by_station = {}
 for header in headers:
 station = header.station_name
 by_station[station] = by_station.get(station, 0) + 1

 print("\nBy Station:")
 for station, count in sorted(by_station.items()):
 print(f" {station}: {count}")

 # Use it
track_operation_usage("ICT", days=30)
```

## Cached Access Pattern

```
class ProcessCache:
 """Wrapper for cached process access with fallback"""

 def __init__(self, api):
 self.api = api
 self._cache = {}

 def get_operation(self, code_or_name):
 """Get operation with local cache layer"""

 # Check local cache first
 if code_or_name in self._cache:
 return self._cache[code_or_name]

 # Get from API (uses API's cache)
 operation = self.api.process.get_operation(code_or_name)

 # Store in local cache
 if operation:
 self._cache[code_or_name] = operation
 self._cache[operation.code] = operation
 self._cache[operation.name] = operation

 return operation

 def clear_cache(self):
 """Clear local cache"""
 self._cache.clear()

Use it
cache = ProcessCache(api)

First access - loads from API
op1 = cache.get_operation("ICT")

Second access - uses local cache
op2 = cache.get_operation("ICT")

Clear when needed
cache.clear_cache()
```

# Practical Examples

## Operation Type Filtering with Validation

```
from typing import List, Optional, Dict, Any
from pywats.exceptions import PyWATSError

def get_valid_operation_code(
 api,
 operation_type: str,
 code_or_name: str
) -> Optional[int]:
 """
 Get a validated operation code for the specified type.

 Useful for report submission where you need to ensure
 the operation code matches the expected type.

 Args:
 api: pyWATS API instance
 operation_type: "test", "repair", or "wip"
 code_or_name: Operation code (int) or name (str)

 Returns:
 Valid operation code or None if invalid

 Example:
 >>> code = get_valid_operation_code(api, "test", "ICT")
 >>> if code:
 ... report.process_code = code
 """
 type_methods = {
 "test": api.process.get_test_operation,
 "repair": api.process.get_repair_operation,
 "wip": api.process.get_wip_operation
 }

 get_method = type_methods.get(operation_type.lower())
 if not get_method:
 raise ValueError(f"Invalid operation_type: {operation_type}. Use 'test', 'repair', or 'wip'")

 # Handle int or str input
 identifier = int(code_or_name) if str(code_or_name).isdigit() else code_or_name

 operation = get_method(identifier)
 return operation.code if operation else None

 def list_operations_by_type(api) -> Dict[str, List[Dict[str, Any]]]:
 """
 Group all operations by their type for easy reference.

 Returns dict like:
 {

```

```

 "test": [{"code": 100, "name": "ICT"}, ...],
 "repair": [{"code": 500, "name": "Rework"}, ...],
 "wip": [{"code": 200, "name": "Assembly"}, ...]
 }
"""

return {
 "test": [
 {"code": op.code, "name": op.name}
 for op in api.process.get_test_operations()
],
 "repair": [
 {"code": op.code, "name": op.name}
 for op in api.process.get_repair_operations()
],
 "wip": [
 {"code": op.code, "name": op.name}
 for op in api.process.get_wip_operations()
]
}

Usage
ops_by_type = list_operations_by_type(api)
print("Available Test Operations:")
for op in ops_by_type["test"]:
 print(f" {op['code']}: {op['name']}")

Validate before report submission
code = get_valid_operation_code(api, "test", "ICT")
if code:
 print(f"Using operation code {code} for test report")
else:
 print("Warning: ICT operation not found, using default")
 code = api.process.get_default_test_code()

```

## Smart Process Cache with Prefetch

```
from datetime import datetime, timedelta
from typing import Dict, Optional, Set
import threading

class SmartProcessCache:
 """
 Enhanced process cache with prefetch and usage tracking.

 Features:
 - Prefetches commonly used operations on init
 - Tracks usage for analytics
 - Thread-safe with automatic refresh
 - Configurable stale threshold
 """

 def __init__(self, api, stale_minutes: int = 5):
 self.api = api
 self.stale_threshold = timedelta(minutes=stale_minutes)
 self._cache: Dict[str, tuple] = {} # key -> (operation, timestamp)
 self._usage_count: Dict[str, int] = {}
 self._lock = threading.Lock()
 self._prefetch_common()

 def _prefetch_common(self) -> None:
 """Prefetch common operation types"""
 common_codes = [100, 500] # Default test and repair
 common_names = ["ICT", "FCT", "Rework", "Assembly"]

 for code in common_codes:
 self._fetch_and_cache(code)
 for name in common_names:
 self._fetch_and_cache(name)

 def _fetch_and_cache(self, identifier) -> Optional[object]:
 """Fetch operation and update cache"""
 op = self.api.process.get_process(identifier)
 if op:
 now = datetime.now()
 with self._lock:
 # Cache by code and name
 self._cache[str(op.code)] = (op, now)
 self._cache[op.name.lower()] = (op, now)
 return op

 def _is_stale(self, timestamp: datetime) -> bool:
 """Check if cached entry is stale"""
 return datetime.now() - timestamp > self.stale_threshold

 def get(self, code_or_name) -> Optional[object]:
 """
 Get operation with smart caching.
 """

Args:
```

```

 code_or_name: Operation code (int) or name (str)

 Returns:
 ProcessInfo or None
 """
key = str(code_or_name).lower() if isinstance(code_or_name, str) else str(code_or_name)

 with self._lock:
 if key in self._cache:
 op, timestamp = self._cache[key]

 # Track usage
 self._usage_count[key] = self._usage_count.get(key, 0) + 1

 # Return if fresh
 if not self._is_stale(timestamp):
 return op

 # Refresh stale or missing entry
 return self._fetch_and_cache(code_or_name)

def get_usage_stats(self) -> Dict[str, int]:
 """Get operation usage statistics"""
 with self._lock:
 return dict(sorted(
 self._usage_count.items(),
 key=lambda x: x[1],
 reverse=True
))

def refresh_all(self) -> None:
 """Force refresh of all cached operations"""
 self.api.process.refresh()
 with self._lock:
 keys = list(self._cache.keys())
 for key in keys:
 self._fetch_and_cache(key)

Usage
cache = SmartProcessCache(api, stale_minutes=10)

Fast cached access
op = cache.get("ICT") # Uses cache if fresh
op = cache.get(100) # Also cached

Check what's being used most
print("Operation usage stats:")
for code, count in cache.get_usage_stats().items():
 print(f" {code}: {count} lookups")

```

## Test Workflow with Operation Validation

```
from dataclasses import dataclass
from typing import Optional
from enum import Enum

class WorkflowStage(Enum):
 """Standard test workflow stages"""
 ICT = "ICT" # In-Circuit Test
 FCT = "FCT" # Functional Test
 BURN_IN = "BURN_IN" # Burn-in / Stress Test
 FVT = "FVT" # Final Verification Test
 REPAIR = "REPAIR" # Repair/Rework
 PACK = "PACK" # Packaging

@dataclass
class WorkflowConfig:
 """Configuration for a test workflow"""
 stages: list # List of WorkflowStage
 operation_codes: dict # Stage -> operation code mapping

def build_workflow(api, stages: list) -> WorkflowConfig:
 """
 Build a validated workflow configuration.

 Ensures all specified stages have valid operation codes
 configured in the WATS system.

 Args:
 api: pyWATS API instance
 stages: List of WorkflowStage values

 Returns:
 WorkflowConfig with validated operation codes

 Raises:
 ValueError: If any stage doesn't have a matching operation
 """
 operation_codes = {}
 missing = []

 for stage in stages:
 # Try to find matching operation
 if stage == WorkflowStage.REPAIR:
 op = api.process.get_repair_operation(stage.value)
 else:
 op = api.process.get_test_operation(stage.value)

 if op:
 operation_codes[stage] = op.code
 else:
 # Try by code for common defaults
 if stage == WorkflowStage.ICT:
 operation_codes[stage] = 100 # Common ICT code
 elif stage == WorkflowStage.REPAIR:
```

```

 operation_codes[stage] = 500 # Common repair code
 else:
 missing.append(stage.value)

if missing:
 available = [p.name for p in api.process.get_processes()]
 raise ValueError(
 f"Missing operations for stages: {missing}. "
 f"Available: {available}"
)

return WorkflowConfig(stages=stages, operation_codes=operation_codes)

def run_workflow_stage(
 api,
 workflow: WorkflowConfig,
 stage: WorkflowStage,
 serial_number: str
) -> dict:
 """
 Execute a workflow stage and return result.

 Args:
 api: pyWATS API instance
 workflow: Validated WorkflowConfig
 stage: Stage to execute
 serial_number: Unit under test serial number

 Returns:
 Dict with stage execution details
 """
 if stage not in workflow.stages:
 raise ValueError(f"Stage {stage} not in workflow")

 operation_code = workflow.operation_codes[stage]
 operation = api.process.get_process_by_code(operation_code)

 return {
 "stage": stage.value,
 "operation_code": operation_code,
 "operation_name": operation.name if operation else "Unknown",
 "serial_number": serial_number,
 "ready": True
 }

Usage: Define and validate a workflow
workflow = build_workflow(api, [
 WorkflowStage.ICT,
 WorkflowStage.FCT,
 WorkflowStage.FVT
])

print("Workflow Configuration:")
for stage, code in workflow.operation_codes.items():
 op = api.process.get_process_by_code(code)
 print(f" {stage.value}: Code {code} ({op.name if op else 'N/A'}))
```

```
Execute stage
result = run_workflow_stage(api, workflow, WorkflowStage.UNIT, "UNIT-001")
print(f"\nReady to run {result['operation_name']} on {result['serial_number']}")
```

## API Reference

### ProcessService Methods

#### Operation Queries

- `get_operation(code_or_name) → Optional[ProcessInfo]` - Get operation by code or name
- `get_test_operation(code_or_name) → Optional[ProcessInfo]` - Get test operation
- `get_repair_operation(code_or_name) → Optional[ProcessInfo]` - Get repair operation
- `get_all_operations() → List[ProcessInfo]` - Get all operations (from cache)

#### Cache Management

- `refresh() → None` - Force cache refresh
- `refresh_interval → int` - Get/set refresh interval in seconds
- `last_refresh → datetime` - Get last refresh timestamp

### Models

#### ProcessInfo

- `id : int` - Process ID
- `name : str` - Operation name
- `code : str` - Short code
- `process_type : str` - Type (TEST, REPAIR, etc.)
- `description : str` - Operation description

### Cache Behavior

- **Default Refresh:** 300 seconds (5 minutes)
- **Auto-Refresh:** When cache age exceeds refresh\_interval
- **Thread-Safe:** Uses threading.Lock for concurrent access
- **Read-Only:** Process definitions are read-only via API

## Best Practices

---

1. **Use the cache** - Don't bypass caching for frequent lookups
  2. **Set appropriate interval** - Balance freshness vs performance
  3. **Lookup by code** - Codes are more stable than names
  4. **Validate codes** - Check operation exists before using
  5. **Refresh on startup** - Ensure cache is fresh when application starts
  6. **Monitor cache age** - Track when last refresh occurred
  7. **Use in reports** - Reference operation codes in test reports
  8. **Handle missing operations** - Gracefully handle unknown codes
- 

## See Also

---

- Report Domain - Use operation codes in test reports
  - Production Domain - Track operations performed on units
  - Analytics Domain - Analyze data by operation type
- 
- 

Source: *docs/modules/product.md*

## Product Domain

---

The Product domain manages product catalog data including product definitions, revisions, bills of materials (BOM), product groups, vendors, and box build templates. This domain is essential for defining WHAT you manufacture before you can track units in production.

## Table of Contents

---

- Quick Start
- Core Concepts
- Product Operations
- Revision Management
- Bill of Materials (BOM)
- Box Build Templates
- Product Groups

- Tags and Metadata
  - Vendors
  - Product Categories
  - Advanced Usage
  - API Reference
- 

## Quick Start

---

### Synchronous Usage

```
from pywats import pyWATS

Initialize the API
api = pyWATS(
 base_url="https://your-wats-server.com",
 token="your-api-token"
)

Get all products
products = api.product.get_products()
for product in products:
 print(f"{product.part_number}: {product.name}")

Get a specific product with full details
product = api.product.get_product("WIDGET-001")
print(f"Product: {product.name}")
print(f"State: {product.state}")

Create a new product
new_product = api.product.create_product(
 part_number="WIDGET-002",
 name="Advanced Widget",
 description="Next generation widget with AI capabilities"
)

Create a revision
revision = api.product.create_revision(
 part_number="WIDGET-002",
 revision="A",
 name="Initial Release",
 description="First production version"
)
```

### Asynchronous Usage

For concurrent requests and better performance:

```

import asyncio
from pywats import AsyncWATS

async def manage_products():
 async with AsyncWATS(
 base_url="https://your-wats-server.com",
 token="your-api-token"
) as api:
 # Get product and its revisions concurrently
 product, revisions = await asyncio.gather(
 api.product.get_product("WIDGET-001"),
 api.product.get_revisions("WIDGET-001")
)

 print(f"Product: {product.name}")
 print(f"Revisions: {len(revisions)}")

asyncio.run(manage_products())

```

## Core Concepts

---

### Product

A **Product** represents a manufactured item in your catalog. It's identified by a unique part number and can have multiple revisions.

#### Key attributes:

- part\_number : Unique identifier (e.g., "WIDGET-001")
- name : Human-readable product name
- description : Detailed description
- state : ACTIVE or INACTIVE
- non\_serial : If True, units don't have serial numbers

### Product Revision

A **ProductRevision** represents a specific version or iteration of a product. Different revisions might have different components, specifications, or manufacturing processes.

#### Key attributes:

- part\_number : Parent product identifier
- revision : Revision identifier (e.g., "A", "1.0", "rev2")
- name : Revision name
- description : What changed in this revision
- state : ACTIVE or INACTIVE

## Product View

A simplified view of a product containing only essential fields. Used for listings where full detail isn't needed.

## Product Operations

### List All Products

```
Get simplified product views (faster, less data)
products = api.product.get_products()
for product in products:
 print(f"{product.part_number}: {product.name} [{product.state}]")

Get full product details (includes all fields)
products_full = api.product.get_products_full()
for product in products_full:
 print(f"{product.part_number}")
 print(f" ID: {product.product_id}")
 print(f" Created: {product.created}")
 print(f" Modified: {product.modified}")
```

### Get Active Products Only

```
Filter for active products only
active_products = api.product.get_active_products()
print(f"Found {len(active_products)} active products")
```

### Get Specific Product

```
Get product by part number
product = api.product.get_product("WIDGET-001")

if product:
 print(f"Product: {product.name}")
 print(f"Part Number: {product.part_number}")
 print(f"State: {product.state}")
 print(f"Non-Serial: {product.non_serial}")
 print(f"Current Revision: {product.revision}")
else:
 print("Product not found")
```

## Create New Product

```
from pywats.domains.product import ProductState

Create a basic product
new_product = api.product.create_product(
 part_number="MAIN-BOARD-001",
 name="Main Circuit Board",
 description="Primary control board for XYZ system",
 state=ProductState.ACTIVE
)

Create a non-serial product (consumables, bulk items)
consumable = api.product.create_product(
 part_number="SCREW-M3-10MM",
 name="M3 x 10mm Screw",
 description="Stainless steel machine screw",
 non_serial=True, # No serial number tracking
 state=ProductState.ACTIVE
)

Create with custom XML data for key-value storage
product_with_metadata = api.product.create_product(
 part_number="WIDGET-003",
 name="Special Widget",
 xml_data='<data><color>blue</color><weight>1.5kg</weight></data>'
)
```

## Update Existing Product

```
Get the product
product = api.product.get_product("WIDGET-001")

Modify fields
product.name = "Updated Widget Name"
product.description = "New and improved description"
product.state = ProductState.ACTIVE

Save changes
updated_product = api.product.update_product(product)
print(f"Updated: {updated_product.part_number}")
```

## Bulk Create/Update Products

```
from pywats.domains.product import Product

Create multiple products at once
products = [
 Product(
 part_number="BATCH-001",
 name="Batch Product 1",
 state=ProductState.ACTIVE
),
 Product(
 part_number="BATCH-002",
 name="Batch Product 2",
 state=ProductState.ACTIVE
),
 Product(
 part_number="BATCH-003",
 name="Batch Product 3",
 state=ProductState.ACTIVE
)
]

Bulk save
saved_products = api.product.bulk_save_products(products)
print(f"Created {len(saved_products)} products")
```

## Check Product State

```
product = api.product.get_product("WIDGET-001")

Check if product is active
if api.product.is_active(product):
 print("Product is active and can be used in production")
else:
 print("Product is inactive")
```

# Revision Management

## List Product Revisions

```
Get all revisions for a product
revisions = api.product.get_revisions("WIDGET-001")

print("Revision History:")
for rev in sorted(revisions, key=lambda r: r.revision):
 print(f" {rev.revision}: {rev.description or 'No description'}")
 if rev.effective_date:
 print(f" Effective: {rev.effective_date}")
```

## Get Specific Revision

```
Get a specific revision
revision = api.product.get_revision("WIDGET-001", "B")

if revision:
 print(f"Revision: {revision.revision}")
 print(f"Name: {revision.name}")
 print(f"Description: {revision.description}")
 print(f"State: {revision.state}")
 print(f"Created: {revision.created}")
```

## Create New Revision

```
Create a new revision
new_revision = api.product.create_revision(
 part_number="WIDGET-001",
 revision="C",
 name="Cost Reduction",
 description="Redesigned for lower manufacturing cost",
 state=ProductState.ACTIVE
)

print(f"Created revision: {new_revision.revision}")
```

## Update Revision

```
Get the revision
revision = api.product.get_revision("WIDGET-001", "B")

Modify
revision.description = "Updated description"
revision.name = "Updated Name"

Save
updated_revision = api.product.update_revision(revision)
```

## Bulk Create/Update Revisions

```
from pywats.domains.product import ProductRevision

Get the product first to get product_id
product = api.product.get_product("WIDGET-001")

Create multiple revisions
revisions = [
 ProductRevision(
 part_number="WIDGET-001",
 revision="D",
 name="Rev D",
 description="Feature enhancement",
 product_id=product.product_id
),
 ProductRevision(
 part_number="WIDGET-001",
 revision="E",
 name="Rev E",
 description="Bug fix",
 product_id=product.product_id
)
]

Bulk save
saved_revisions = api.product.bulk_save_revisions(revisions)
print(f"Created {len(saved_revisions)} revisions")
```

## Bill of Materials (BOM)

The BOM defines the components and raw materials needed to manufacture a product revision.

## Get BOM

```
Get BOM as raw WSBF XML string
bom_xml = api.product.get_bom("WIDGET-001", "A")
if bom_xml:
 print(f"BOM XML length: {len(bom_xml)} characters")

Get BOM as structured list of items
bom_items = api.product.get_bom_items("WIDGET-001", "A")

print(f"Bill of Materials for WIDGET-001 Rev A:")
for item in bom_items:
 print(f" {item.part_number} Rev {item.revision}")
 print(f" Quantity: {item.quantity}")
 print(f" Reference: {item.reference_designator or 'N/A'}")
 print(f" Description: {item.description or 'N/A'}")
```

## Update BOM

```
from pywats.domains.product import BomItem

Define BOM items
bom_items = [
 BomItem(
 part_number="RESISTOR-10K",
 revision="A",
 quantity=5,
 reference_designator="R1,R2,R3,R4,R5",
 description="10K ohm resistor"
),
 BomItem(
 part_number="CAPACITOR-100UF",
 revision="A",
 quantity=2,
 reference_designator="C1,C2",
 description="100uF electrolytic capacitor"
),
 BomItem(
 part_number="IC-MICROCONTROLLER",
 revision="B",
 quantity=1,
 reference_designator="U1",
 description="Main microcontroller"
)
]

Update the BOM
success = api.product.update_bom(
 part_number="WIDGET-001",
 revision="A",
 bom_items=bom_items,
 description="Updated BOM for cost reduction"
)

if success:
 print("BOM updated successfully")
```

## Box Build Templates

### ⚠ INTERNAL API - Subject to change

Box build templates define PRODUCT-LEVEL relationships - what subunits are REQUIRED to build a product. This is different from production-level assembly (which is in the Production domain).

#### Use Cases:

- Define product structure (parent-child relationships)

- Specify which subassemblies make up a final product
- Track component quantities needed

**Important:** This is a DESIGN-TIME definition. To attach actual physical units during production, use

```
api.production.add_child_to_assembly()
```

## Get Box Build Template

```
Get or create a box build template
template = api.product.get_box_build_template("MAIN-ASSEMBLY", "A")

print(f"Box Build Template for {template.part_number} Rev {template.revision}")
print(f"Current subunits: {len(template.subunits)})")
```

## Add Subunits to Template

```
Get the template
template = api.product.get_box_build_template("MAIN-ASSEMBLY", "A")

Add subunits (components that make up this product)
template.add_subunit("PCBA-001", "A", quantity=1)
template.add_subunit("PSU-MODULE", "B", quantity=1)
template.add_subunit("ENCLOSURE", "A", quantity=1)
template.add_subunit("CABLE-ASSY", "C", quantity=3)

Save changes to server
template.save()
print(f"Template updated with {len(template.subunits)} subunits")
```

## List Template Subunits

```
template = api.product.get_box_build_template("MAIN-ASSEMBLY", "A")

print("Required subunits:")
for subunit in template.subunits:
 print(f" {subunit.child_part_number} Rev {subunit.child_revision}")
 print(f" Quantity: {subunit.quantity}")
```

## Update Subunit Quantity

```
template = api.product.get_box_build_template("MAIN-ASSEMBLY", "A")

Update quantity for a subunit
template.update_subunit("CABLE-ASSY", "C", quantity=5) # Changed from 3 to 5

Save changes
template.save()
```

## Remove Subunit from Template

```
template = api.product.get_box_build_template("MAIN-ASSEMBLY", "A")

Remove a subunit
template.remove_subunit("CABLE-ASSY", "C")

Save changes
template.save()
```

## Complete Box Build Example

```
Define a complete product structure
template = api.product.get_box_build_template("LAPTOP-X1", "A")

Add all required components
template.add_subunit("MOTHERBOARD", "A", quantity=1)
template.add_subunit("SCREEN-15INCH", "B", quantity=1)
template.add_subunit("KEYBOARD-US", "A", quantity=1)
template.add_subunit("BATTERY-PACK", "C", quantity=1)
template.add_subunit("RAM-8GB", "A", quantity=2) # 2 x 8GB modules
template.add_subunit("SSD-512GB", "A", quantity=1)

Save all at once
template.save()

print(f'Laptop structure defined with {len(template.subunits)} components')
```

## Product Groups

Product groups allow you to organize products into logical categories for reporting and filtering.

## List All Product Groups

```
Get all product groups
groups = api.product.get_groups()

print("Product Groups:")
for group in groups:
 print(f" {group.name}: {group.description or 'No description'}")
```

## Filter Product Groups

```
Get groups with OData filter
groups = api.product.get_groups(
 filter_str="contains(name,'Electronics')",
 top=10
)
```

## Get Groups for Specific Product

```
Get all groups that include this product
groups = api.product.get_groups_for_product("WIDGET-001", "A")

print(f"Product WIDGET-001 Rev A belongs to {len(groups)} groups:")
for group in groups:
 print(f" - {group.name}")
```

## Tags and Metadata

Tags provide key-value metadata storage for products and revisions. Useful for custom attributes, classifications, or integration data.

## Product Tags

```
Get all tags for a product
tags = api.product.get_product_tags("WIDGET-001")

print("Product Tags:")
for tag in tags:
 print(f" {tag['key']}: {tag['value']}")

Add a single tag
api.product.add_product_tag(
 part_number="WIDGET-001",
 key="ManufacturingLocation",
 value="Factory-A"
)

Set multiple tags at once (replaces all existing tags)
new_tags = [
 {"key": "Category", "value": "Electronics"},
 {"key": "RoHS", "value": "Compliant"},
 {"key": "LeadTime", "value": "2-weeks"}
]

api.product.set_product_tags("WIDGET-001", new_tags)
```

## Revision Tags

```
Get tags for a specific revision
tags = api.product.get_revision_tags("WIDGET-001", "B")

print("Revision Tags:")
for tag in tags:
 print(f" {tag['key']}: {tag['value']}")

Add a revision-specific tag
api.product.add_revision_tag(
 part_number="WIDGET-001",
 revision="B",
 key="ECO",
 value="ECO-2024-0542"
)

Set multiple revision tags
revision_tags = [
 {"key": "ApprovalDate", "value": "2024-06-15"},
 {"key": "ApprovedBy", "value": "J.Smith"}
]

api.product.set_revision_tags("WIDGET-001", "B", revision_tags)
```

# Vendors

Manage vendor information for procurement and supply chain tracking.

## List Vendors

```
Get all vendors
vendors = api.product.get_vendors()

print("Vendors:")
for vendor in vendors:
 print(f" {vendor.get('name', 'Unknown')}")
 print(f" Contact: {vendor.get('contact', 'N/A')}")
 print(f" Email: {vendor.get('email', 'N/A')}
```

## Create/Update Vendor

```
Create a new vendor
vendor_data = {
 "name": "Acme Components Inc.",
 "contact": "John Doe",
 "email": "sales@acmecomponents.com",
 "phone": "+1-555-0123",
 "address": "123 Industrial Pkwy, Tech City, TC 12345"
}

saved_vendor = api.product.save_vendor(vendor_data)
print(f"Created vendor: {saved_vendor.get('name')}")

Update existing vendor (include vendor ID)
vendor_data = {
 "vendorId": "existing-vendor-id",
 "name": "Acme Components Inc.",
 "contact": "Jane Smith", # Updated contact
 "email": "newsales@acmecomponents.com" # Updated email
}

updated_vendor = api.product.save_vendor(vendor_data)
```

## Delete Vendor

```
Delete a vendor
success = api.product.delete_vendor("vendor-id-to-delete")

if success:
 print("Vendor deleted")
```

# Product Categories

---

## INTERNAL API - Subject to change

Categories provide another way to classify products.

## Get Product Categories

```
Get all categories
categories = api.product.get_product_categories()

print("Product Categories:")
for category in categories:
 print(f" {category.get('name', 'Unknown')}")
 print(f" ID: {category.get('id')}")
```

## Assign Categories to Product

```
Set categories for a product (by category IDs)
category_ids = [
 "cat-id-1",
 "cat-id-2",
 "cat-id-3"
]

api.product.save_product_categories("WIDGET-001", category_ids)
print("Categories assigned to product")
```

# Advanced Usage

## Complete Product Lifecycle

```
from pywats.domains.product import ProductState

1. Create the product
product = api.product.create_product(
 part_number="NEW-WIDGET",
 name="Next Generation Widget",
 description="Revolutionary new design",
 state=ProductState.ACTIVE
)

2. Create initial revision
revision = api.product.create_revision(
 part_number="NEW-WIDGET",
 revision="A",
 name="Prototype",
 description="Initial prototype version",
 state=ProductState.ACTIVE
)

3. Define BOM
from pywats.domains.product import BomItem

bom = [
 BomItem(part_number="COMPONENT-1", revision="A", quantity=2),
 BomItem(part_number="COMPONENT-2", revision="B", quantity=1),
]

api.product.update_bom("NEW-WIDGET", "A", bom)

4. Add to product group
groups = api.product.get_groups(filter_str="name eq 'New Products'")
(Groups are managed via WATS UI or separate API)

5. Add metadata tags
api.product.add_product_tag("NEW-WIDGET", "Status", "In Development")
api.product.add_revision_tag("NEW-WIDGET", "A", "TestPhase", "Alpha")

print("Product lifecycle setup complete!")
```

## Product Family Management

```
Define a product family with common base
base_products = ["WIDGET-BASE", "WIDGET-PLUS", "WIDGET-PRO"]

for pn in base_products:
 # Check if product exists
 existing = api.product.get_product(pn)

 if not existing:
 # Create if doesn't exist
 product = api.product.create_product(
 part_number=pn,
 name=f"{pn} Product",
 state=ProductState.ACTIVE
)

 # Create initial revision
 api.product.create_revision(
 part_number=pn,
 revision="A",
 name="Initial Release"
)

 # Tag with family membership
 api.product.add_product_tag(pn, "ProductFamily", "Widget Series")

 print(f"Created {pn}")
 else:
 print(f"{pn} already exists")
```

## Migration/Import Helper

```
def import_products_from_csv(csv_file_path):
 """Import products from CSV file"""
 import csv

 products = []

 with open(csv_file_path, 'r') as file:
 reader = csv.DictReader(file)
 for row in reader:
 product = api.product.create_product(
 part_number=row['part_number'],
 name=row['name'],
 description=row.get('description', ''),
 state=ProductState.ACTIVE
)

 if product:
 # Create default revision
 api.product.create_revision(
 part_number=row['part_number'],
 revision=row.get('revision', 'A'),
 name=row.get('revision_name', 'Initial')
)

 products.append(product)

 return products

Use it
imported = import_products_from_csv('products.csv')
print(f"Imported {len(imported)} products")
```

## API Reference

### ProductService Methods

#### Product Operations

- `get_products() → List[ProductView]` - Get all products as simplified views
- `get_products_full() → List[Product]` - Get all products with full details
- `get_product(part_number) → Optional[Product]` - Get specific product
- `get_products_batch(part_numbers, max_workers=10) → List[Result[Product]]` - Fetch multiple products concurrently
- `create_product(...) → Optional[Product]` - Create new product

- `update_product(product) → Optional[Product]` - Update existing product
- `bulk_save_products(products) → List[Product]` - Bulk create/update
- `is_active(product) → bool` - Check if product is active
- `get_active_products() → List[ProductView]` - Get only active products

## Revision Operations

- `get_revision(part_number, revision) → Optional[ProductRevision]` - Get specific revision
- `get_revisions(part_number) → List[ProductRevision]` - Get all revisions for product
- `get_revisions_batch(pairs, max_workers=10) → List[Result[ProductRevision]]` - Fetch multiple revisions concurrently (pairs = list of (part\_number, revision) tuples)
- `create_revision(...) → Optional[ProductRevision]` - Create new revision
- `update_revision(revision) → Optional[ProductRevision]` - Update existing revision
- `bulk_save_revisions(revisions) → List[ProductRevision]` - Bulk create/update

## BOM Operations

- `get_bom(part_number, revision) → Optional[str]` - Get BOM as WSBF XML
- `get_bom_items(part_number, revision) → List[BomItem]` - Get BOM as structured items
- `update_bom(part_number, revision, bom_items, description) → bool` - Update BOM

## Group Operations

- `get_groups(filter_str, top) → List[ProductGroup]` - Get product groups
- `get_groups_for_product(part_number, revision) → List[ProductGroup]` - Get groups for product

## Tag Operations

- `get_product_tags(part_number) → List[Dict[str, str]]` - Get product tags
- `set_product_tags(part_number, tags) → Optional[Product]` - Set product tags
- `add_product_tag(part_number, key, value) → Optional[Product]` - Add single tag
- `get_revision_tags(part_number, revision) → List[Dict[str, str]]` - Get revision tags
- `set_revision_tags(part_number, revision, tags) → Optional[ProductRevision]` - Set revision tags
- `add_revision_tag(part_number, revision, key, value) → Optional[ProductRevision]` - Add single tag

## Vendor Operations

- `get_vendors() → List[Dict[str, Any]]` - Get all vendors
- `save_vendor(vendor_data) → Optional[Dict[str, Any]]` - Create/update vendor
- `delete_vendor(vendor_id) → bool` - Delete vendor

## ProductServiceInternal Methods (⚠️ Subject to change)

### Box Build Operations

- `get_box_build(part_number, revision) → BoxBuildTemplate` - Get/create template
- `get_bom(part_number, revision) → List[BomItem]` - Get BOM items
- `set_bom(part_number, revision, bom_items) → bool` - Set BOM items

### Category Operations

- `get_product_categories() → List[Dict[str, Any]]` - Get all categories
- `set_product_categories(part_number, category_ids) → bool` - Assign categories

## Models

### Product

- `product_id : UUID` (auto-generated)
- `part_number : str` (required, unique)
- `name : Optional[str]`
- `description : Optional[str]`
- `revision : Optional[str]` (current/default revision)
- `state : ProductState` (ACTIVE/INACTIVE)
- `non_serial : bool` (default False)
- `xml_data : Optional[str]` (custom metadata)
- `created : Optional[datetime]`
- `modified : Optional[datetime]`

### ProductRevision

- `product_revision_id : UUID` (auto-generated)
- `product_id : UUID` (links to Product)
- `part_number : str`
- `revision : str` (required)
- `name : Optional[str]`
- `description : Optional[str]`
- `state : ProductState`
- `effective_date : Optional[datetime]`
- `xml_data : Optional[str]`
- `created : Optional[datetime]`

- `modified` : `Optional[datetime]`

## BomItem

- `part_number` : `str`
- `revision` : `str`
- `quantity` : `int`
- `reference_designator` : `Optional[str]`
- `description` : `Optional[str]`

## ProductView

- `part_number` : `str`
- `name` : `Optional[str]`
- `non_serial` : `Optional[bool]`
- `state` : `Optional[ProductState]`

## ProductGroup

- `product_group_id` : `UUID`
  - `name` : `str`
  - `description` : `Optional[str]`
- 

## Best Practices

1. **Always create a revision** - Even for the first version, create revision "A" or "1.0"
2. **Use meaningful part numbers** - Use a consistent naming scheme (e.g., "WIDGET-001", "PCB-MAIN-V2")
3. **Set product state** - Mark obsolete products as INACTIVE rather than deleting them
4. **Use tags for custom attributes** - Instead of modifying the database schema, use tags
5. **Bulk operations for efficiency** - When importing or updating many products, use bulk methods
6. **BOM accuracy** - Keep BOMs up-to-date as product revisions change
7. **Box Build vs BOM** - Use BOM for component lists, Box Build for structural relationships
8. **Document revisions** - Always include meaningful descriptions when creating revisions
9. **Non-serial products** - Mark consumables, bulk items, or non-tracked items as `non_serial=True`
10. **Version control** - Use revision tags to track ECOs, approval dates, and change history

## Common Workflows

### New Product Introduction (NPI)

```
1. Create product definition
product = api.product.create_product(
 part_number="NEW-PRODUCT",
 name="New Product Name",
 description="Product description"
)

2. Create engineering sample revision
api.product.create_revision(
 part_number="NEW-PRODUCT",
 revision="ES",
 name="Engineering Sample",
 description="Pre-production samples"
)

3. Tag as NPI
api.product.add_product_tag("NEW-PRODUCT", "Phase", "NPI")
api.product.add_revision_tag("NEW-PRODUCT", "ES", "Status", "Engineering Validation")

4. When ready for production, create production revision
production_rev = api.product.create_revision(
 part_number="NEW-PRODUCT",
 revision="A",
 name="Production Release",
 description="First production version"
)

5. Update tags
api.product.add_product_tag("NEW-PRODUCT", "Phase", "Production")
api.product.add_revision_tag("NEW-PRODUCT", "A", "Status", "Released")
```

### Product Obsolescence

```
Mark product as inactive (don't delete - preserve history)
product = api.product.get_product("OLD-WIDGET")
product.state = ProductState.INACTIVE
api.product.update_product(product)

Tag with obsolescence info
api.product.add_product_tag("OLD-WIDGET", "Obsolete", "true")
api.product.add_product_tag("OLD-WIDGET", "ObsoleteDate", "2024-12-31")
api.product.add_product_tag("OLD-WIDGET", "Replacement", "NEW-WIDGET")
```

# Troubleshooting

---

## Product not found

```
product = api.product.get_product("WIDGET-001")
if not product:
 print("Product doesn't exist - create it first")
```

## Revision creation fails

```
Make sure the product exists first
product = api.product.get_product("WIDGET-001")
if not product:
 print("Create the product before creating revisions")
else:
 revision = api.product.create_revision("WIDGET-001", "A", ...)
```

## BOM update fails

```
Ensure the product revision exists
revision = api.product.get_revision("WIDGET-001", "A")
if not revision:
 print("Create the product revision first")
else:
 api.product.update_bom("WIDGET-001", "A", bom_items)
```

---

## See Also

---

- Production Domain - Managing units in production
- Report Domain - Querying test results
- Asset Domain - Managing test equipment and tools

# Production Domain

The Production domain manages units (individual physical products) throughout their manufacturing lifecycle. It tracks unit creation, phases, processes, verification status, assembly relationships, and change history. This domain is where you track ACTUAL units being manufactured, from creation to finalization.

## **IMPORTANT: Server Feature Requirement**

The Production module is an optional feature that **must be enabled on the WATS server by the WATS team**. If production tracking is not enabled on your server, this module will not work and API calls will fail.

Contact your WATS administrator or the WATS support team to verify that production tracking is enabled for your server before using this module.

## Table of Contents

- Quick Start
- Core Concepts
- Unit Operations
- Unit Verification
- Phase Management
- Process Tracking
- Assembly Management
- Serial Number Allocation
- Unit Changes
- Unit Phases (Internal)
- Advanced Usage
- API Reference

# Quick Start

---

## Synchronous Usage

```
from pywats import pyWATS

Initialize the API
api = pyWATS(
 base_url="https://your-wats-server.com",
 token="your-api-token"
)

Get a production unit
unit = api.production.get_unit(
 serial_number="WIDGET-12345",
 part_number="WIDGET-001"
)

print(f"Unit: {unit.serial_number}")
print(f"Part: {unit.part_number} Rev {unit.part_revision}")
print(f"Status: {unit.status}")
print(f"Current Phase: {unit.current_phase}")
print(f"Current Process: {unit.current_process}")

Check if unit is passing all tests
if api.production.is_unit_passing("WIDGET-12345", "WIDGET-001"):
 print("✓ Unit is passing all tests")

Move to finalized phase
api.production.set_unit_phase(
 serial_number="WIDGET-12345",
 part_number="WIDGET-001",
 phase="Finalized"
)
```

## Asynchronous Usage

For concurrent requests and better performance:

```
import asyncio
from pywats import AsyncWATS

async def check_units():
 async with AsyncWATS(
 base_url="https://your-wats-server.com",
 token="your-api-token"
) as api:
 # Check multiple units concurrently
 serials = ["SN-001", "SN-002", "SN-003"]
 units = await asyncio.gather(*[
 api.production.get_unit(sn, "WIDGET-001")
 for sn in serials
])

 for unit in units:
 print(f"{unit.serial_number}: {unit.status}")

asyncio.run(check_units())
```

```
else:
print("X Unit has test failures")
```

# Create a new unit

---

```
from pywats.domains.production import Unit

new_unit = Unit(
 serial_number="WIDGET-12346",
 part_number="WIDGET-001",
 part_revision="B",
 status="In Production"
)

created = api.production.create_units([new_unit])
```

---

## ## Core Concepts

### ### Unit

A \*\*Unit\*\* represents a single physical product being manufactured. Each unit has a unique serial number and is associated with a product/revision.

**Key attributes:**

- `serial\_number`: Unique identifier for this specific unit
- `part\_number`: Product being manufactured
- `part\_revision`: Product revision
- `status`: Current status (e.g., "In Production", "Tested", "Shipped")
- `current\_phase`: Current workflow phase (e.g., "ICT", "FCT", "Finalized")
- `current\_process`: Current operation type code (e.g., 10 for ICT, 50 for FCT)

### ### Unit Lifecycle

Units typically follow this lifecycle:

1. **Creation** - Unit is created in the system
2. **In Production** - Unit enters manufacturing
3. **Testing** - Unit goes through test operations (ICT, FCT, etc.)
4. **Verification** - Test results are verified (Pass/Fail)
5. **Repair** (if failed) - Failed units go to repair
6. **Assembly** (if applicable) - Subunits are attached
7. **Finalized** - Unit passes all requirements and is finalized
8. **Shipped** - Unit is shipped to customer

### ### Unit Phase

A \*\*Unit Phase\*\* represents a stage in the production workflow. Phases are configured in WATS and can include:

- **In Test** - Unit is currently being tested
- **Passed** - Unit passed the current test
- **Failed** - Unit failed the current test
- **In Repair** - Unit is being repaired
- **Finalized** - Unit completed all requirements
- **Scrapped** - Unit was scrapped

Phases can be referenced by:

- **ID**: Numeric identifier (e.g., 123)
- **Code**: Phase code string (e.g., "ICT\_PASSED")
- **Name**: Display name (e.g., "ICT - Passed")

### ### Unit Verification

**UnitVerification** provides test result status:

**Grades:**

- `PASSED`: All tests passed
- `FAILED`: Some tests failed
- `NOT\_TESTED`: Unit hasn't been tested yet
- `INCOMPLETE`: Testing in progress

### ### Assembly

Units can have **parent-child relationships** to represent product assemblies:

- A \*\*Main Assembly\*\* unit can have multiple \*\*Subunit\*\* children
- Each child is attached at a specific \*\*position\*\* or slot
- Children must be \*\*finalized\*\* before attachment
- Assembly structure must match the \*\*Box Build Template\*\* (defined in Product domain)

#### \*\*Key distinction:\*\*

- \*\*Box Build Template\*\* (Product domain): Defines WHAT subunits are REQUIRED (design)
- \*\*Unit Assembly\*\* (Production domain): Attaches ACTUAL units (production)

---

## ## Unit Operations

### ### Get Specific Unit

```
```python
# Get unit by serial number and part number
unit = api.production.get_unit(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001"
)

if unit:
    print(f"Serial: {unit.serial_number}")
    print(f"Part: {unit.part_number} Rev {unit.part_revision}")
    print(f"Status: {unit.status}")
    print(f"Phase: {unit.current_phase}")
    print(f"Process: {unit.current_process}")
    print(f"Created: {unit.created}")
    print(f"Modified: {unit.modified}")

# Check if unit has Product info attached
if unit.product:
    print(f"Product Name: {unit.product.name}")

if unit.product_revision:
    print(f"Revision Name: {unit.product_revision.name}")
```

Create Units

```
from pywats.domains.production import Unit
from datetime import datetime

# Create single unit
unit = Unit(
    serial_number="WIDGET-NEW-001",
    part_number="WIDGET-001",
    part_revision="C",
    status="Created"
)

created = api.production.create_units([unit])
print(f"Created: {created[0].serial_number}")

# Create multiple units in batch
units = [
    Unit(
        serial_number=f"BATCH-{i:04d}",
        part_number="WIDGET-001",
        part_revision="C",
        status="Created",
        created=datetime.now()
    )
    for i in range(1, 11) # Create 10 units
]

created_units = api.production.create_units(units)
print(f"Created {len(created_units)} units")
```

Update Unit

```
# Get unit
unit = api.production.get_unit("WIDGET-12345", "WIDGET-001")

# Modify fields
unit.status = "Testing"
unit.location = "Test Station 3"

# Save changes
updated = api.production.update_unit(unit)
print(f"Updated: {updated.serial_number}")
```

Unit Verification

Check if a unit is passing or failing tests.

Verify Unit Status

```
from pywats.domains.production import UnitVerificationGrade

# Get detailed verification
verification = api.production.verify_unit(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001",
    revision="B"
)

if verification:
    print(f"Verification Grade: {verification.grade}")
    print(f"Is Passing: {verification.is_passing}")
    print(f"Failed Steps: {verification.failed_step_count}")
    print(f"Passed Steps: {verification.passed_step_count}")

# Check specific grade
if verification.grade == UnitVerificationGrade.PASSED:
    print("✓ All tests passed")
elif verification.grade == UnitVerificationGrade.FAILED:
    print("✗ Some tests failed")
elif verification.grade == UnitVerificationGrade.NOT_TESTED:
    print("⚠ Unit not yet tested")
elif verification.grade == UnitVerificationGrade.INCOMPLETE:
    print("⏳ Testing in progress")
```

Get Unit Grade

```
# Get just the grade (simpler than full verification)
grade = api.production.get_unit_grade(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001"
)

if grade == UnitVerificationGrade.PASSED:
    print("Ready for next phase")
```

Simple Pass/Fail Check

```
# Quick boolean check
if api.production.is_unit_passing("WIDGET-12345", "WIDGET-001"):
    print("✓ Unit is passing - can proceed")

    # Finalize the unit
    api.production.set_unit_phase(
        serial_number="WIDGET-12345",
        part_number="WIDGET-001",
        phase="Finalized"
    )
else:
    print("✗ Unit has failures - needs repair")

    # Send to repair
    api.production.set_unit_phase(
        serial_number="WIDGET-12345",
        part_number="WIDGET-001",
        phase="In Repair"
)
```

Phase Management

Manage unit workflow phases.

Get All Phases

```
# Get all available phases
phases = api.production.get_phases()

print("Available Phases:")
for phase in phases:
    print(f"  ID: {phase.unit_phase_id}")
    print(f"  Code: {phase.code}")
    print(f"  Name: {phase.name}")
    print(f"  Flag: {phase.phase_flags}")
    print()
```

Get Specific Phase

```
# By phase ID
phase = api.production.get_phase(123)

# By phase code
phase = api.production.get_phase("ICT_PASSED")

# By phase name
phase = api.production.get_phase("ICT - Passed")

if phase:
    print(f"Found phase: {phase.name} (ID: {phase.unit_phase_id})")
```

Set Unit Phase

```
# Set phase by name
api.production.set_unit_phase(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001",
    phase="ICT",
    comment="Starting ICT test"
)

# Set phase by ID
api.production.set_unit_phase(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001",
    phase=123, # Phase ID
    comment="Moving to next phase"
)

# Set phase using enum
from pywats.domains.production import UnitPhaseFlag

api.production.set_unit_phase(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001",
    phase=UnitPhaseFlag.FINALIZED,
    comment="Unit completed successfully"
)
```

Process Tracking

Track which test operation a unit is currently at.

Set Unit Process

```
# Set process by operation type code
# (Operation types are defined in the Process domain)

# Set to ICT (operation type 10)
api.production.set_unit_process(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001",
    process_code=10,  # ICT
    comment="Starting In-Circuit Test"
)

# Set to FCT (operation type 50)
api.production.set_unit_process(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001",
    process_code=50,  # Final Function Test
    comment="Starting Final Test"
)

# Clear process (set to None)
api.production.set_unit_process(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001",
    process_code=None,
    comment="Process completed"
)
```

Assembly Management

Manage parent-child unit relationships for box builds.

Add Child to Assembly

```
# Prerequisites:  
# 1. Box build template must define the child as valid (Product domain)  
# 2. Child unit must be finalized  
# 3. Child cannot already have a parent  
  
# Add a child unit to parent assembly  
success = api.production.add_child_to_assembly(  
    parent_serial="MODULE-001",  
    parent_part="MAIN-MODULE",  
    child_serial="PCBA-12345",  
    child_part="PCBA-BOARD"  
)  
  
if success:  
    print("Child added to assembly")
```

Remove Child from Assembly

```
# Remove a child unit from parent  
success = api.production.remove_child_from_assembly(  
    parent_serial="MODULE-001",  
    parent_part="MAIN-MODULE",  
    child_serial="PCBA-12345",  
    child_part="PCBA-BOARD"  
)  
  
if success:  
    print("Child removed from assembly")
```

Verify Assembly

```
# Check if assembly is complete (all required children attached)  
verification = api.production.verify_assembly(  
    serial_number="MODULE-001",  
    part_number="MAIN-MODULE",  
    revision="A"  
)  
  
if verification:  
    print(f"Assembly Complete: {verification.get('is_complete', False)}")  
  
    if not verification.get('is_complete'):  
        missing = verification.get('missing_children', [])  
        print(f"Missing {len(missing)} required children:")  
        for child in missing:  
            print(f" - {child.get('part_number')}")
```

Complete Assembly Workflow

```
# 1. Define what subunits are required (Product domain)
template = api.product.get_box_build_template("MAIN-MODULE", "A")
template.add_subunit("PCBA-BOARD", "A", quantity=1)
template.add_subunit("POWER-SUPPLY", "B", quantity=1)
template.add_subunit("CABLE-ASSY", "A", quantity=2)
template.save()

# 2. Create parent unit
from pywats.domains.production import Unit

parent = Unit(
    serial_number="MODULE-001",
    part_number="MAIN-MODULE",
    part_revision="A"
)
api.production.create_units([parent])

# 3. Create and finalize child units
pcba = Unit(serial_number="PCBA-001", part_number="PCBA-BOARD", part_revision="A")
psu = Unit(serial_number="PSU-001", part_number="POWER-SUPPLY", part_revision="B")
cable1 = Unit(serial_number="CABLE-001", part_number="CABLE-ASSY", part_revision="A")
cable2 = Unit(serial_number="CABLE-002", part_number="CABLE-ASSY", part_revision="A")

api.production.create_units([pcba, psu, cable1, cable2])

# Test and finalize children (simplified - would actually run tests)
for child in [pcba, psu, cable1, cable2]:
    api.production.set_unit_phase(
        child.serial_number,
        child.part_number,
        phase="Finalized"
)

# 4. Attach children to parent
api.production.add_child_to_assembly("MODULE-001", "MAIN-MODULE", "PCBA-001", "PCBA-BOARD")
api.production.add_child_to_assembly("MODULE-001", "MAIN-MODULE", "PSU-001", "POWER-SUPPLY")
api.production.add_child_to_assembly("MODULE-001", "MAIN-MODULE", "CABLE-001", "CABLE-ASSY")
api.production.add_child_to_assembly("MODULE-001", "MAIN-MODULE", "CABLE-002", "CABLE-ASSY")

# 5. Verify assembly is complete
verification = api.production.verify_assembly("MODULE-001", "MAIN-MODULE", "A")

if verification.get('is_complete'):
    print("✓ Assembly complete - all required children attached")

    # Finalize the parent
    api.production.set_unit_phase("MODULE-001", "MAIN-MODULE", phase="Finalized")
else:
    print("✗ Assembly incomplete")
```

Serial Number Allocation

Allocate serial numbers from configured sequences.

Get Serial Number Types

```
# Get all serial number types
sn_types = api.production.get_serial_number_types()

print("Serial Number Types:")
for sn_type in sn_types:
    print(f"  {sn_type.name}")
    print(f"    Pattern: {sn_type.pattern}")
    print(f"    Next: {sn_type.next_number}")
```

Allocate Serial Numbers

```
# Allocate a single serial number
serial_numbers = api.production.allocate_serial_numbers(
    type_name="Production_SN",
    quantity=1
)

if serial_numbers:
    new_sn = serial_numbers[0]
    print(f"Allocated: {new_sn}")

# Allocate multiple serial numbers
batch_sns = api.production.allocate_serial_numbers(
    type_name="Production_SN",
    quantity=100
)

print(f"Allocated {len(batch_sns)} serial numbers:")
print(f"  First: {batch_sns[0]}")
print(f"  Last: {batch_sns[-1]})

# Use allocated serial numbers to create units
from pywats.domains.production import Unit

units = [
    Unit(
        serial_number=sn,
        part_number="WIDGET-001",
        part_revision="B"
    )
    for sn in batch_sns
]

api.production.create_units(units)
```

Unit Changes

Track changes to unit phase and status.

Get Unit Changes

```
# Get all changes for a specific unit
changes = api.production.get_unit_changes(
    serial_number="WIDGET-12345",
    part_number="WIDGET-001"
)

print(f"Unit change history ({len(changes)} changes):")
for change in changes:
    print(f"  {change.changed}: Phase {change.old_unit_phase_id} → {change.new_unit_phase_id}")
    print(f"    User: {change.user or 'System'}")
    print(f"    Comment: {change.comment or 'No comment'}")

# Get recent changes (all units, last 50)
recent_changes = api.production.get_unit_changes(top=50)

print(f"Recent changes across all units:")
for change in recent_changes[:10]:
    print(f"  {change.serial_number}: {change.comment or 'Phase change'}")
```

Acknowledge Unit Change

```
# Mark a change as acknowledged
change_id = "some-change-uuid"

success = api.production.acknowledge_unit_change(change_id)

if success:
    print("Change acknowledged")
```

Unit Phases (Internal)

INTERNAL API - Subject to change

Get unit phase definitions from internal API.

Get Unit Phases

```
# Get all unit phases from internal API
phases = api.production.get_all_unit_phases()

print("Unit Phases (Internal API):")
for phase in phases:
    print(f"  {phase.name} (ID: {phase.unit_phase_id})")
    print(f"    Code: {phase.code}")
    print(f"    Flags: {phase.phase_flags}")
    print(f"    Description: {phase.description or 'N/A'}")
```

Advanced Usage

Complete Production Workflow

```
def production_workflow(serial_number, part_number, revision):
    """Complete workflow from creation to finalization"""

    # 1. Create unit
    from pywats.domains.production import Unit

    unit = Unit(
        serial_number=serial_number,
        part_number=part_number,
        part_revision=revision,
        status="Created"
    )
    api.production.create_units([unit])
    print(f"1. Created unit: {serial_number}")

    # 2. Start production - Set to ICT
    api.production.set_unit_phase(
        serial_number, part_number,
        phase="Under Production"
    )

    api.production.set_unit_process(
        serial_number, part_number,
        process_code=10, # ICT
        comment="Starting In-Circuit Test"
    )
    print("2. Set to ICT process")

    # 3. Run ICT test (this would create a UUT report)
    # ... test execution code ...
    # api.report.submit_report(ict_report)
    print("3. ICT test completed")

    # 4. Check if passed
    if api.production.is_unit_passing(serial_number, part_number):
        print("4. ICT PASSED")

        # Move to FCT
        api.production.set_unit_process(
            serial_number, part_number,
            process_code=50, # FCT
            comment="Starting Final Function Test"
        )
        print("5. Set to FCT process")

        # Run FCT test
        # ... test execution code ...
        # api.report.submit_report(fct_report)
        print("6. FCT test completed")
```

```
# Check final result
if api.production.is_unit_passing(serial_number, part_number):
    print("7. FCT PASSED")

    # Finalize unit
    api.production.set_unit_phase(
        serial_number, part_number,
        phase="Finalized",
        comment="All tests passed successfully"
    )
    print("8. Unit FINALIZED ✓")
    return True

else:
    print("7. FCT FAILED")
    # Send to repair
    api.production.set_unit_phase(
        serial_number, part_number,
        phase="In Repair"
    )
    return False

else:
    print("4. ICT FAILED")
    # Send to repair
    api.production.set_unit_phase(
        serial_number, part_number,
        phase="In Repair"
    )
    return False

# Use it
production_workflow("WIDGET-12345", "WIDGET-001", "B")
```

Repair Workflow

```
def repair_workflow(serial_number, part_number):
    """Handle failed unit repair and retest"""

    # 1. Check current status
    verification = api.production.verify_unit(serial_number, part_number)

    if verification.is_passing:
        print("Unit is already passing - no repair needed")
        return True

    print(f"Unit has {verification.failed_step_count} failed steps")

    # 2. Set to repair phase
    api.production.set_unit_phase(
        serial_number, part_number,
        phase="In Repair",
        comment="Starting repair"
    )

    # 3. Create repair report (UUR - Unit Under Repair)
    # This would typically be done by creating a UUR report
    # See Report domain documentation for details

    # 4. Set back to test process
    unit = api.production.get_unit(serial_number, part_number)

    api.production.set_unit_process(
        serial_number, part_number,
        process_code=unit.current_process, # Return to failed operation
        comment="Re-testing after repair"
    )

    # 5. Run test again
    # ... test execution code ...

    # 6. Check if now passing
    if api.production.is_unit_passing(serial_number, part_number):
        print("✓ Repair successful - unit now passing")

        # Move to next phase
        api.production.set_unit_phase(
            serial_number, part_number,
            phase="Passed",
            comment="Passed after repair"
        )
        return True
    else:
        print("✗ Still failing after repair - escalate")

        # Could scrap or send to advanced repair
        api.production.set_unit_phase(
            serial_number, part_number,
            phase="Failed",
```

```
        comment="Failed after repair attempt"
    )
    return False

# Use it
repair_workflow("WIDGET-12345", "WIDGET-001")
```

Batch Production Tracking

```
def batch_production(part_number, revision, lot_number, quantity=100):
    """Create and track a production batch"""
    from pywats.domains.production import Unit
    from datetime import datetime

    # 1. Allocate serial numbers
    serial_numbers = api.production.allocate_serial_numbers(
        type_name="Production_SN",
        quantity=quantity
    )

    print(f"Allocated {quantity} serial numbers")
    print(f"  Range: {serial_numbers[0]} to {serial_numbers[-1]}")

    # 2. Create units
    units = [
        Unit(
            serial_number=sn,
            part_number=part_number,
            part_revision=revision,
            status="Created",
            lot_number=lot_number  # If your Unit model has this field
        )
        for sn in serial_numbers
    ]

    created = api.production.create_units(units)
    print(f"Created {len(created)} units in lot {lot_number}")

    # 3. Track batch progress
    batch_stats = {
        'created': len(created),
        'tested': 0,
        'passed': 0,
        'failed': 0,
        'in_repair': 0,
        'finalized': 0
    }

    # After testing, update stats
    for sn in serial_numbers:
        unit = api.production.get_unit(sn, part_number)

        if unit.current_phase:
            if "Finalized" in unit.current_phase:
                batch_stats['finalized'] += 1
            elif "Repair" in unit.current_phase:
                batch_stats['in_repair'] += 1
            elif "Passed" in unit.current_phase:
                batch_stats['passed'] += 1
            elif "Failed" in unit.current_phase:
                batch_stats['failed'] += 1
```

```
print("\nBatch Statistics:")
print(f"  Created: {batch_stats['created']}")
print(f"  Finalized: {batch_stats['finalized']}")
print(f"  Passed: {batch_stats['passed']}")
print(f"  Failed: {batch_stats['failed']}")
print(f"  In Repair: {batch_stats['in_repair']}")

yield_pct = (batch_stats['finalized'] / batch_stats['created']) * 100
print(f"\nYield: {yield_pct:.1f}%")

return batch_stats

# Use it
stats = batch_production("WIDGET-001", "C", "LOT-2025-W01", quantity=50)
```

Phase Gate Checks

```
def can_advance_to_next_phase(serial_number, part_number, target_phase):
    """Check if unit can advance to next phase"""

    # Get current unit state
    unit = api.production.get_unit(serial_number, part_number)

    if not unit:
        print("Unit not found")
        return False

    # Check verification
    verification = api.production.verify_unit(serial_number, part_number)

    if not verification:
        print("Cannot verify unit")
        return False

    # Define phase gate rules
    if target_phase == "Finalized":
        # Must be passing all tests
        if not verification.is_passing:
            print(f"X Cannot finalize - unit has {verification.failed_step_count} failures")
            return False

        # Must not be in repair
        if unit.current_phase and "Repair" in unit.current_phase:
            print("X Cannot finalize - unit is in repair")
            return False

        print("✓ Unit can be finalized")
        return True

    elif target_phase == "Shipped":
        # Must be finalized first
        if not unit.current_phase or "Finalized" not in unit.current_phase:
            print("X Cannot ship - unit not finalized")
            return False

        print("✓ Unit can be shipped")
        return True

    # Add more phase gate rules as needed
    return True

# Use it
if can_advance_to_next_phase("WIDGET-12345", "WIDGET-001", "Finalized"):
    api.production.set_unit_phase("WIDGET-12345", "WIDGET-001", "Finalized")
```

API Reference

ProductionService Methods

Unit Operations

- `get_unit(serial_number, part_number) → Optional[Unit]` - Get unit details
- `create_units(units) → List[Unit]` - Create one or more units
- `update_unit(unit) → Optional[Unit]` - Update existing unit

Unit Verification

- `verify_unit(serial_number, part_number, revision) → Optional[UnitVerification]` - Get verification details
- `get_unit_grade(serial_number, part_number, revision) → Optional[UnitVerificationGrade]` - Get grade only
- `is_unit_passing(serial_number, part_number) → bool` - Quick pass/fail check

Unit Phases

- `get_phases() → List[UnitPhase]` - Get all available phases
- `get_phase(identifier) → Optional[UnitPhase]` - Get specific phase by ID/code/name
- `get_phase_id(phase) → Optional[int]` - Get phase ID from identifier
- `set_unit_phase(serial_number, part_number, phase, comment) → bool` - Set unit phase
- `set_unit_process(serial_number, part_number, process_code, comment) → bool` - Set current process

Unit Changes

- `get_unit_changes(serial_number, part_number, top) → List[UnitChange]` - Get change history
- `acknowledge_unit_change(change_id) → bool` - Acknowledge a change

Assembly Operations

- `add_child_to_assembly(parent_serial, parent_part, child_serial, child_part) → bool` - Add child unit
- `remove_child_from_assembly(parent_serial, parent_part, child_serial, child_part) → bool` - Remove child unit
- `verify_assembly(serial_number, part_number, revision) → Optional[Dict]` - Verify assembly completeness

Serial Number Operations

- `get_serial_number_types() → List[SerialNumberType]` - Get SN types
- `allocate_serial_numbers(type_name, quantity) → List[str]` - Allocate serial numbers

ProductionServiceInternal Methods (⚠️ Subject to change)

Unit Phase Operations

- `get_unit_phases() → List[UnitPhase]` - Get unit phases from internal API

Models

Unit

- `serial_number : str` (required)
- `part_number : str` (required)
- `part_revision : Optional[str]`
- `status : Optional[str]`
- `current_phase : Optional[str]`
- `current_process : Optional[int]`
- `product : Optional[Product]` (attached)
- `product_revision : Optional[ProductRevision]` (attached)
- `created : Optional[datetime]`
- `modified : Optional[datetime]`

UnitVerification

- `grade : UnitVerificationGrade`
- `is_passing : bool`
- `failed_step_count : int`
- `passed_step_count : int`

UnitVerificationGrade (Enum)

- `PASSED` : All tests passed
- `FAILED` : Some tests failed
- `NOT_TESTED` : Not yet tested
- `INCOMPLETE` : Testing in progress

UnitPhase

- unit_phase_id : int
- code : str
- name : str
- phase_flags : Optional[int]
- description : Optional[str]

UnitPhaseFlag (Enum)

- Numeric values for common phases
- Example: UnitPhaseFlag.FINALIZED

UnitChange

- unit_change_id : UUID
- serial_number : str
- part_number : str
- old_unit_phase_id : Optional[int]
- new_unit_phase_id : Optional[int]
- changed : datetime
- user : Optional[str]
- comment : Optional[str]

SerialNumberType

- name : str
- pattern : str
- next_number : int

Best Practices

1. **Always verify before finalizing** - Check `is_unit_passing()` before setting phase to "Finalized"
2. **Use meaningful comments** - Add comments when changing phases or processes
3. **Track batch context** - Use lot numbers or batch IDs to group related units
4. **Finalize children first** - Children must be finalized before assembly attachment
5. **Verify assemblies** - Always call `verify_assembly()` before finalizing box builds

- 6. Handle repair workflow** - Set proper phase when sending units to repair
 - 7. Track changes** - Use `get_unit_changes()` to audit unit lifecycle
 - 8. Allocate SNs from sequences** - Use configured SN types instead of manual generation
 - 9. Phase gates** - Implement validation before phase transitions
 - 10. Don't skip phases** - Follow your defined production flow sequence
-

Common Workflows

Test Station Integration

```
def run_test_at_station(serial_number, part_number, station, operation_type):  
    """Execute test and update unit status"""  
  
    # Set unit to current process  
    api.production.set_unit_process(  
        serial_number, part_number,  
        process_code=operation_type,  
        comment=f"Testing at {station}"  
    )  
  
    # Run test (creates UUT report)  
    # This is typically done by test equipment/software  
    # report = create_test_report(serial_number, operation_type)  
    # api.report.submit_report(report)  
  
    # After test, check result  
    if api.production.is_unit_passing(serial_number, part_number):  
        # Move to next phase  
        api.production.set_unit_phase(  
            serial_number, part_number,  
            phase="Passed",  
            comment=f"Passed at {station}"  
        )  
        return True  
    else:  
        # Send to repair  
        api.production.set_unit_phase(  
            serial_number, part_number,  
            phase="Failed",  
            comment=f"Failed at {station}"  
        )  
        return False
```

Multi-Station Production Line

```
def production_line(serial_number, part_number):
    """Move unit through multiple stations"""

    stations = [
        {"name": "ICT", "process": 10},
        {"name": "FCT", "process": 50},
        {"name": "Burn-In", "process": 60},
        {"name": "Final Inspection", "process": 100}
    ]

    for station in stations:
        print(f"\n== {station['name']} ==")

        # Set process
        api.production.set_unit_process(
            serial_number, part_number,
            process_code=station['process'],
            comment=f"At {station['name']}"
        )

        # Simulate test execution
        # In real scenario, this calls test software
        time.sleep(1) # Placeholder for actual test

        # Check result
        if api.production.is_unit_passing(serial_number, part_number):
            print("✓ PASS at {station['name']} ")
        else:
            print("✗ FAIL at {station['name']} ")
            return False

    # All stations passed - finalize
    api.production.set_unit_phase(
        serial_number, part_number,
        phase="Finalized",
        comment="Completed all production stages"
    )

    print("\n✓ Unit FINALIZED")
    return True
```

Troubleshooting

Unit not found

```
unit = api.production.get_unit("WIDGET-12345", "WIDGET-001")
if not unit:
    print("Unit doesn't exist - create it first")
```

Cannot finalize unit

```
# Check if passing
if not api.production.is_unit_passing("WIDGET-12345", "WIDGET-001"):
    verification = api.production.verify_unit("WIDGET-12345", "WIDGET-001")
    print(f"Cannot finalize - {verification.failed_step_count} failures")
```

Cannot add child to assembly

```
# Common reasons:
# 1. Child not finalized
api.production.set_unit_phase(child_sn, child_pn, "Finalized")

# 2. Child not in box build template
template = api.product.get_box_build_template(parent_pn, parent_rev)
template.add_subunit(child_pn, child_rev).save()

# 3. Child already has a parent
# Remove from old parent first
api.production.remove_child_from_assembly(old_parent_sn, old_parent_pn, child_sn, child_pn)
```

See Also

- Product Domain - Defining products and box build templates
- Report Domain - Creating test reports for units
- Process Domain - Defining operation types and workflows
- Analytics Domain - Analyzing unit flow and yield

Analytics Domain

The Analytics domain provides data analysis and visualization capabilities for test results. It includes yield calculations, measurement aggregation, Cpk analysis, and production flow visualization (Unit Flow). Use this domain to understand manufacturing performance, identify trends, and optimize processes.

Table of Contents

- Quick Start
 - Type-Safe Enums
 - Core Concepts
 - Yield Analysis
 - Measurement Aggregation
 - Unit Flow Analysis
 - Advanced Usage
 - API Reference
-

Quick Start

Synchronous Usage

```
from pywats import pyWATS, WATSFilter, StatusFilter, MeasurementPath
from datetime import datetime, timedelta

# Initialize
api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your-api-token"
)

# Get yield for last 7 days (using type-safe enum)
filter_obj = WATSFilter(
    part_number="WIDGET-001",
    status=StatusFilter.ALL, # Type-safe enum
    days=7
)

yield_result = api.analytics.get_yield(filter_obj)
print(f"Yield: {yield_result.yield_pct:.1f}%")
print(f"Passed: {yield_result.passed}")
print(f"Failed: {yield_result.failed}")
print(f"Total: {yield_result.total}")

# Get measurements for a step (using friendly path format)
path = MeasurementPath("Numeric Limit Tests/3.3V Rail")
measurements = api.analytics.get_aggregated_measurements(
    filter_obj,
    measurement_paths=path
)

for meas in measurements:
    print(f"{meas.step_name}:")
    print(f"  Avg: {meas.avg:.3f}")
    print(f"  Cpk: {meas.cpk:.2f}")
```

Asynchronous Usage

For concurrent requests and better performance:

```
import asyncio
from pywats import AsyncWATS, WATSFilter, StatusFilter

async def analyze_yield():
    async with AsyncWATS(
        base_url="https://your-wats-server.com",
        token="your-api-token"
    ) as api:
        filter_obj = WATSFilter(part_number="WIDGET-001", days=7)

        # Concurrent queries - much faster!
        yield_result, measurements = await asyncio.gather(
            api.analytics.get_yield(filter_obj),
            api.analytics.get_aggregated_measurements(filter_obj)
        )

        print(f"Yield: {yield_result.yield_pct:.1f}%")

asyncio.run(analyze_yield())
```

Type-Safe Enums

The Analytics domain provides type-safe enums for building queries with IDE autocomplete and compile-time validation.

Query Enums

```
from pywats import StatusFilter, RunFilter, StepType, CompOperator

# Status filter - filter by test result status
StatusFilter.PASSED      # "P" - Passed tests only
StatusFilter.FAILED       # "F" - Failed tests only
StatusFilter.ERROR        # "E" - Error/terminated tests
StatusFilter.TERMINATED   # "T" - Terminated tests
StatusFilter.ALL          # "*" - All statuses

# Run filter - filter by test run sequence
RunFilter.FIRST          # 1 - First run only
RunFilter.LAST           # -1 - Last run only
RunFilter.ALL            # 0 - All runs
RunFilter.ALL_EXCEPT_FIRST # 2 - All except first run

# Step types returned in analytics results
StepType.NUMERIC_LIMIT   # Numeric measurement with limits
StepType.STRING_VALUE     # String comparison
StepType.PASS_FAIL        # Binary pass/fail
StepType.SEQUENCECALL     # Sequence call step

# Comparison operators in step results
CompOperator.GELE         # Greater/Equal and Less/Equal (within range)
CompOperator.GT             # Greater than
CompOperator.LT             # Less than
CompOperator.EQ             # Equal
CompOperator.NE             # Not equal
CompOperator.LOG            # Logged value (no comparison)
```

Dimension and KPI Enums

```
from pywats import Dimension, KPI, RepairDimension, RepairKPI

# Grouping dimensions for dynamic queries
Dimension.PART_NUMBER      # Group by part number
Dimension.STATION_NAME       # Group by test station
Dimension.PERIOD              # Group by time period
Dimension.PROCESS_CODE        # Group by process code
Dimension.REVISION             # Group by product revision
Dimension.OPERATOR             # Group by operator
Dimension.FIXTURE_ID          # Group by fixture
# ... and 15+ more dimensions

# KPIs for yield/performance queries
KPI.UNIT_COUNT      # Number of units tested
KPI.FPY                # First Pass Yield
KPI.RTY                # Rolled Throughput Yield
KPI.PPM_FPY            # Parts per million (FPY based)
KPI.AVG_TEST_TIME      # Average test time
KPI.RETEST_COUNT        # Number of retests
# ... and 15+ more KPIs

# Repair-specific dimensions and KPIs
RepairDimension.REPAIR_CODE      # Repair code
RepairDimension.COMPONENT_REF     # Component reference
RepairKPI.REPAIR_COUNT           # Number of repairs
RepairKPI.REPAIR_REPORT_COUNT    # Repair reports
```

DimensionBuilder - Fluent API

Build dimension queries with type safety and IDE autocomplete:

```
from pywats import DimensionBuilder, Dimension, KPI, WATSFilter

# Build custom dimension query
dims = DimensionBuilder()\
    .add(KPI.UNIT_COUNT, desc=True) \
    .add(KPI.FPY) \
    .add(Dimension.PART_NUMBER) \
    .add(Dimension.PERIOD) \
    .build()
# Result: "unitCount desc;fpY;partNumber;period"

# Use with WATSFilter
filter_obj = WATSFilter(
    part_number="WIDGET-001",
    dimensions=dims,
    period_count=30
)

# Use presets for common patterns
dims = DimensionBuilder.yield_by_product().build()
dims = DimensionBuilder.yield_by_station().build()
dims = DimensionBuilder.repair_analysis().build()
dims = DimensionBuilder.oee_analysis().build()
```

Path Utilities

Work with measurement paths using a user-friendly format:

```

from pywats import MeasurementPath, StepPath

# Create paths with forward slashes (user-friendly format)
path = MeasurementPath("Functional Test/Power Supply/3.3V Rail")
print(path.display)      # Functional Test/Power Supply/3.3V Rail
print(path.api_format)   # Functional Test\Power Supply\3.3V Rail (internal)
print(path.parts)        # ['Functional Test', 'Power Supply', '3.3V Rail']

# Build paths with concatenation
parent = StepPath("Functional Test")
full_path = parent / "Power Supply" / "3.3V Rail"
print(full_path.display) # Functional Test/Power Supply/3.3V Rail

# Use from_parts for programmatic construction
path = MeasurementPath.from_parts(["Main", "SubTest", "Measurement"])

# Paths work directly in API calls
measurements = api.analytics.get_aggregated_measurements(
    filter_obj,
    measurement_paths=path # Automatic conversion to API format
)

# Access display format from results
for meas in measurements:
    print(f"Path: {meas.step_path_display}") # Friendly format

```

Core Concepts

Yield

Yield is the percentage of units that passed testing:

- `yield_pct` : Percentage (0-100)
- `passed` : Number of passed units
- `failed` : Number of failed units
- `total` : Total units tested

Measurements

Aggregated measurements provide statistical analysis of numeric test steps:

- `avg` : Average value
- `min` , `max` : Minimum and maximum
- `std_dev` : Standard deviation
- `cpk` : Process capability index
- `count` : Number of measurements

Unit Flow

Unit Flow visualizes how units move through production:

- **Nodes**: Operations/processes (e.g., "ICT", "FCT", "Assembly")
- **Links**: Transitions between operations
- **Units**: Individual units tracked through the flow

Yield Analysis

Get Overall Yield

```
from pywats import WATSFilter, StatusFilter

# Yield for last 30 days
filter_obj = WATSFilter(
    part_number="WIDGET-001",
    status=StatusFilter.ALL, # Include all statuses
    days=30
)

yield_result = api.analytics.get_yield(filter_obj)

print(f"== YIELD ANALYSIS ==")
print(f"Yield: {yield_result.yield_pct:.2f}%")
print(f"Passed: {yield_result.passed}")
print(f"Failed: {yield_result.failed}")
print(f"Total: {yield_result.total}")
```

Yield by Date Range

```
from datetime import datetime, timedelta

# Last quarter
end_date = datetime.now()
start_date = end_date - timedelta(days=90)

filter_obj = WATSFilter(
    part_number="WIDGET-001",
    start_date_time=start_date,
    end_date_time=end_date
)

yield_result = api.analytics.get_yield(filter_obj)
print(f"Q1 Yield: {yield_result.yield_pct:.1f}%")
```

Yield by Station

```
# Compare yield across stations
stations = ["ICT-01", "ICT-02", "ICT-03"]

print("== YIELD BY STATION ==")
for station in stations:
    filter_obj = WATSFilter(
        part_number="WIDGET-001",
        station=station,
        days=7
    )

    yield_result = api.analytics.get_yield(filter_obj)
    print(f"{station}: {yield_result.yield_pct:.1f}% ({yield_result.total} units)")
```

Yield Trend Over Time

```
from datetime import datetime, timedelta

def get_daily_yield(part_number, days=30):
    """Get yield for each of the last N days"""

    trends = []
    end_date = datetime.now()

    for i in range(days):
        day_end = end_date - timedelta(days=i)
        day_start = day_end - timedelta(days=1)

        filter_obj = WATSFilter(
            part_number=part_number,
            start_date_time=day_start,
            end_date_time=day_end
        )

        yield_result = api.analytics.get_yield(filter_obj)

        trends.append({
            'date': day_start.date(),
            'yield': yield_result.yield_pct,
            'total': yield_result.total
        })

    return trends

# Use it
trends = get_daily_yield("WIDGET-001", days=14)

print("== 14-DAY YIELD TREND ==")
for trend in reversed(trends):
    yield_bar = "█" * int(trend['yield'] / 5) # Visual bar
    print(f"{trend['date']}: {trend['yield']:.1f}% {yield_bar} ({trend['total']} units)")
```

Measurement Aggregation

Get Measurement Statistics

```
from pywats import WATSFilter, MeasurementPath

# Get measurements for a specific test step
filter_obj = WATSFilter(
    part_number="WIDGET-001",
    days=7
)

# Use user-friendly path format (slashes)
path = MeasurementPath("Power Supply Tests/3.3V Rail")
measurements = api.analytics.get_aggregated_measurements(
    filter_obj,
    measurement_paths=path
)

if measurements:
    meas = measurements[0]
    print(f"Step: {meas.step_name}")
    print(f"Count: {meas.count}")
    print(f"Average: {meas.avg:.3f}")
    print(f"Min: {meas.min:.3f}")
    print(f"Max: {meas.max:.3f}")
    print(f"Std Dev: {meas.std_dev:.3f}")
    print(f"Cpk: {meas.cpk:.2f}")
```

Multiple Measurements

```
from pywats import MeasurementPath

# Get multiple measurements at once using path objects
step_paths = [
    MeasurementPath("Power Supply Tests/3.3V Rail"),
    MeasurementPath("Power Supply Tests/5V Rail"),
    MeasurementPath("Power Supply Tests/12V Rail")
]

measurements = api.analytics.get_aggregated_measurements(
    filter_obj,
    measurement_paths=step_paths
)

print("== POWER RAIL MEASUREMENTS ==")
for meas in measurements:
    print(f"\n{meas.step_name}:")
    print(f"  Path: {meas.step_path_display}") # User-friendly format
    print(f"  Avg: {meas.avg:.3f} (\u03c3={meas.std_dev:.3f})")
    print(f"  Range: {meas.min:.3f} to {meas.max:.3f}")
    print(f"  Cpk: {meas.cpk:.2f}")
```

Cpk Analysis

```
from pywats import MeasurementPath

def analyze_process_capability(part_number, step_path, days=30):
    """Analyze process capability (Cpk) for a measurement"""

    filter_obj = WATSFilter(
        part_number=part_number,
        days=days
    )

    # Accept string or MeasurementPath
    path = MeasurementPath(step_path) if isinstance(step_path, str) else step_path

    measurements = api.analytics.get_aggregated_measurements(
        filter_obj,
        measurement_paths=path
    )

    if not measurements:
        print("No measurements found")
        return

    meas = measurements[0]

    print(f"\n== PROCESS CAPABILITY: {meas.step_name} ==")
    print(f"Cpk: {meas.cpk:.2f}")

    if meas.cpk >= 1.33:
        print("✓ Process is capable (Cpk ≥ 1.33)")
    elif meas.cpk >= 1.0:
        print("△ Process is marginally capable (1.0 ≤ Cpk < 1.33)")
    else:
        print("✗ Process is not capable (Cpk < 1.0)")

    print(f"\nStatistics:")
    print(f"  Mean: {meas.avg:.3f}")
    print(f"  Std Dev: {meas.std_dev:.3f}")
    print(f"  Range: {meas.min:.3f} to {meas.max:.3f}")
    print(f"  Samples: {meas.count}")

# Use it with user-friendly path
analyze_process_capability("WIDGET-001", "Power Supply/3.3V Rail", days=90)
```

Unit Flow Analysis (Internal)

INTERNAL API - Subject to change

Unit Flow visualizes production flow and identifies bottlenecks.

Basic Unit Flow Query

```
from pywats.domains.analytics import UnitFlowFilter
from datetime import datetime, timedelta

# Create filter
flow_filter = UnitFlowFilter(
    part_number="WIDGET-001",
    date_from=datetime.now() - timedelta(days=7),
    date_to=datetime.now()
)

# Get flow data
flow = api.analytics.get_unit_flow(flow_filter)

print(f"== UNIT FLOW ==")
print(f"Nodes: {len(flow.nodes)}")
print(f"Links: {len(flow.links)}")
print(f"Units: {len(flow.units)}")

# Show nodes (operations)
print("\nOperations:")
for node in flow.nodes:
    print(f" {node.name}: {node.unit_count} units")

# Show transitions
print("\nTransitions:")
for link in flow.links:
    print(f" {link.source_name} → {link.target_name}: {link.unit_count} units")
```

Identify Bottlenecks

```
# Get bottlenecks (nodes with high unit count)
flow_filter = UnitFlowFilter(
    part_number="WIDGET-001",
    days=7
)

bottlenecks = api.analytics.get_bottlenecks(flow_filter)

print("== BOTTLENECKS ==")
for bottleneck in bottlenecks:
    print(f"{bottleneck.operation_name}:")
    print(f" Units waiting: {bottleneck.unit_count}")
    print(f" Avg time: {bottleneck.avg_time_hours:.1f} hours")
```

Trace Serial Numbers Through Flow

```
# Trace specific units
flow_filter = UnitFlowFilter(
    part_number="WIDGET-001",
    serial_numbers=["W12345", "W12346", "W12347"],
    days=30
)

trace_result = api.analytics.trace_serial_numbers(flow_filter)

for unit in trace_result.units:
    print(f"\n{unit.serial_number}:")
    print(f"  Status: {unit.status}")
    print(f"  Path: {' → '.join(unit.node_path)}")
    print(f"  Total time: {unit.total_time_hours:.1f} hours")
```

Advanced Usage

Yield Dashboard

```
from pywats import WATSFilter

def yield_dashboard(part_numbers, days=7):
    """Generate yield dashboard for multiple products"""

    print("=" * 70)
    print(f"YIELD DASHBOARD ({days} days)")
    print("=" * 70)

    results = []

    for pn in part_numbers:
        filter_obj = WATSFilter(part_number=pn, days=days)
        yield_result = api.analytics.get_yield(filter_obj)

        results.append({
            'part': pn,
            'yield': yield_result.yield_pct,
            'passed': yield_result.passed,
            'failed': yield_result.failed,
            'total': yield_result.total
        })

    # Sort by yield (ascending)
    results.sort(key=lambda x: x['yield'])

    print(f"\n{'Part Number':<20} {'Yield':>8} {'Passed':>8} {'Failed':>8} {'Total':>8}")
    print("-" * 70)

    for r in results:
        status = "✓" if r['yield'] >= 95 else "⚠" if r['yield'] >= 85 else "✗"
        print(f"{r['part']:<20} {r['yield']:>7.1f}% {r['passed']:>8} {r['failed']:>8} {r['total']:>8}{status}")

    # Summary
    avg_yield = sum(r['yield'] for r in results) / len(results)
    total_units = sum(r['total'] for r in results)

    print("-" * 70)
    print(f"{'AVERAGE':<20} {avg_yield:>7.1f}% {'':>8} {'':>8} {total_units:>8}")
    print("=" * 70)

# Use it
products = ["WIDGET-001", "WIDGET-002", "WIDGET-003", "GADGET-001"]
yield_dashboard(products, days=30)
```

Measurement Report

```
from pywats import WATSFilter, MeasurementPath

def measurement_report(part_number, step_paths, days=7):
    """Generate measurement report for multiple steps"""

    filter_obj = WATSFilter(part_number=part_number, days=days)

    # Convert string paths to MeasurementPath objects
    paths = [MeasurementPath(p) if isinstance(p, str) else p for p in step_paths]

    measurements = api.analytics.get_aggregated_measurements(
        filter_obj,
        measurement_paths=paths
    )

    print(f"\n== MEASUREMENT REPORT: {part_number} ({days} days) ==\n")
    print(f"{'Step':<40} {'Avg':>10} {'Std Dev':>10} {'Cpk':>8} {'Count':>8}")
    print("-" * 80)

    for meas in measurements:
        cpk_status = "✓" if meas.cpk >= 1.33 else "▲" if meas.cpk >= 1.0 else "✗"
        print(f"{meas.step_name:<40} {meas.avg:>10.3f} {meas.std_dev:>10.3f} {meas.cpk:>7.2f}"
            f"{meas.count:>8} {cpk_status}")

    print("=" * 80)

    # Use it with user-friendly paths
    steps = [
        "Power Supply Tests/3.3V Rail",
        "Power Supply Tests/5V Rail",
        "Power Supply Tests/12V Rail",
        "Temperature Tests/Idle Temp",
        "Temperature Tests/Load Temp"
    ]
    measurement_report("WIDGET-001", steps, days=30)
```

API Reference

AnalyticsService Methods

All methods are accessed via `api.analytics`. Methods marked with  use internal WATS endpoints that may change without notice.

Yield Operations

- `get_yield(filter)` → `YieldResult` - Calculate yield statistics

Measurement Operations

- `get_aggregated_measurements(filter, measurement_paths)` → `List[AggregatedMeasurement]` - Get measurement statistics

Unit Flow Operations (⚠ Internal API)

- `get_unit_flow(filter)` → `UnitFlowResult` - Get complete flow data
- `get_flow_nodes()` → `List[UnitFlowNode]` - Get flow nodes
- `get_flow_links()` → `List[UnitFlowLink]` - Get flow links
- `get_flow_units()` → `List[UnitFlowUnit]` - Get flow units
- `get_bottlenecks(filter, min_yield_threshold)` → `List[UnitFlowNode]` - Identify bottlenecks
- `trace_serial_numbers(serial_numbers, filter)` → `UnitFlowResult` - Trace specific units

Step/Measurement Filter Operations (⚠ Internal API)

- `get_measurement_list(filter_data, step_filters, sequence_filters)` → `List[MeasurementListItem]` - Measurement values with XML filters
- `get_measurement_list_by_product(product_group_id, level_id, days, step_filters, sequence_filters)` → `List[MeasurementListItem]` - Simple query
- `get_step_status_list(filter_data, step_filters, sequence_filters)` → `List[StepStatusItem]` - Step statuses with XML filters
- `get_step_status_list_by_product(product_group_id, level_id, days, step_filters, sequence_filters)` → `List[StepStatusItem]` - Simple query
- `get_top_failed_advanced(filter_data, top_count)` → `List[TopFailedStep]` - Top failed steps with advanced filters
- `get_top_failed_by_product(part_number, process_code, product_group_id, level_id, days, count)` → `List[TopFailedStep]` - Simple query

Enums

Query Enums (from `pywats`)

Enum	Values	Description
StatusFilter	PASSED , FAILED , ERROR , TERMINATED , ALL	Filter by test status
RunFilter	FIRST , LAST , ALL , ALL_EXCEPT_FIRST	Filter by run sequence
StepType	NUMERIC_LIMIT , STRING_VALUE , PASS_FAIL , SEQUENCECALL , etc.	Step type classification
CompOperator	GELE , GT , LT , EQ , NE , LOG , etc.	Comparison operators
SortDirection	ASC , DESC	Sort direction for dimension queries

Dimension/KPI Enums (from `pywats` or `pywats.domains.analytics`)

Enum	Count	Description
Dimension	23+	Grouping dimensions (PART_NUMBER, STATION_NAME, PERIOD, etc.)
RepairDimension	15	Repair-specific dimensions
KPI	21+	Key Performance Indicators (FPY, RTY, PPM_FPY, etc.)
RepairKPI	2	Repair KPIs

DimensionBuilder Class Methods

Method	Description
add(dimension, direction=None, asc=False, desc=False)	Add dimension with optional sort
add_all(*dimensions)	Add multiple dimensions
build()	Build semicolon-separated string
clear()	Clear all dimensions
yield_by_product()	Preset for product yield analysis
yield_by_station()	Preset for station yield analysis
repair_analysis()	Preset for repair statistics
oee_analysis()	Preset for OEE metrics

Path Utilities (from pywats)

Class	Description
StepPath	Step path with / ↔ ↴ conversion
MeasurementPath	Measurement path with / ↔ ↴ conversion

Properties: .display (user-friendly), .api_format (internal), .parts (list)

Methods: from_parts(parts) , concatenation with / operator

Models

YieldResult

- yield_pct :float (percentage)
- passed :int
- failed :int
- total :int

AggregatedMeasurement

- step_name :str

- step_path : str (API format with `\`)
- step_path_display : str (user-friendly with `/`)
- count : int
- avg : float
- min , max : float
- std_dev : float
- cpk : float (process capability index)

TopFailedStep

- step_name , step_path : str
- step_path_display : str (user-friendly)
- step_type : Union[StepType, str]
- fail_count , total_count : int
- step_filter , sequence_filter : str (XML for drill-down)

StepAnalysisRow

- step_name , step_path : str
- step_path_display : str (user-friendly)
- step_type : Union[StepType, str]
- comp_operator : Union[CompOperator, str]
- Statistical fields: avg , std_dev , cpk , etc.

UnitFlowFilter

- part_number : str
- date_from , date_to : datetime
- days : int (shortcut for last N days)
- serial_numbers : List[str] (optional filter)

StepStatusItem (⚠ Internal API)

- step_name , step_path , step_type : str - Step identification
- pass_count , fail_count , total_count : int - Status counts
- status : str - Current status
- serial_number , report_id : str - Associated unit info

MeasurementListItem (⚠ Internal API)

- serial_number , part_number , revision : str - Unit identification

- `step_name` , `step_path` : str - Step identification
 - `value` : float - Measured value
 - `limit_low` , `limit_high` : float - Limits
 - `status` : str - Pass/Fail status
 - `unit` : str - Unit of measurement
-

Best Practices

1. **Use type-safe enums** - Leverage `StatusFilter` , `RunFilter` , `Dimension` , `KPI` for IDE autocomplete
 2. **Use path utilities** - `MeasurementPath("Main/SubTest/Meas")` handles format conversion automatically
 3. **Use DimensionBuilder presets** - Start with `DimensionBuilder.yield_by_product()` and customize
 4. **Start with simple queries** - Use default parameters, add filters as needed
 5. **Monitor Cpk trends** - $Cpk < 1.0$ indicates process capability issues
 6. **Investigate anomalies** - Low Cpk or sudden yield drops
 7. **Use Unit Flow for bottlenecks** - Visualize production flow
-

See Also

- Report Domain - Test reports and measurements
 - Production Domain - Unit lifecycle and status
 - Process Domain - Operation type definitions
-

Source: [docs/modules/rootcause.md](#)

RootCause Domain

The RootCause domain provides issue tracking and defect management capabilities. Use this to create tickets for failures, track root cause investigations, assign work, and close issues. It supports status-based workflows (Open → In Progress → Resolved → Closed), priority management, and view filtering (assigned to me, following, all).

Table of Contents

- Quick Start
 - Core Concepts
 - Ticket Management
 - Status and Workflow
 - Priority Management
 - View Filtering
 - Search and Query
 -  Known Issues & Workarounds
 - Advanced Usage
 - API Reference
-

Quick Start

Synchronous Usage

```
from pywats import pyWATS

# Initialize
api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your-api-token"
)

# Get all open tickets
open_tickets = api.rootcause.get_open_tickets()

print(f"== OPEN TICKETS ({len(open_tickets)}) ==")
for ticket in open_tickets:
    print(f"\n#{ticket.id}: {ticket.subject}")
    print(f"  Priority: {ticket.priority}")
    print(f"  Assigned: {ticket.assigned_to}")

# Get specific ticket
ticket = api.rootcause.get_ticket(12345)

if ticket:
    print(f"\nTicket: {ticket.subject}")
    print(f"Status: {ticket.status}")
    print(f"Description: {ticket.description}")

# Get active tickets (open + in progress)
from pywats.domains.rootcause.models import TicketView

active = api.rootcause.get_active_tickets(view=TicketView.ASSIGNED)
print(f"\n{len(active)} tickets assigned to me")
```

Asynchronous Usage

For concurrent requests and better performance:

```

import asyncio
from pywats import AsyncWATS
from pywats.domains.rootcause.models import TicketView

async def track_issues():
    async with AsyncWATS(
        base_url="https://your-wats-server.com",
        token="your-api-token"
    ) as api:
        # Fetch different ticket views concurrently
        my_tickets, all_open = await asyncio.gather(
            api.rootcause.get_active_tickets(view=TicketView.ASSIGNED),
            api.rootcause.get_open_tickets()
        )

        print(f"My tickets: {len(my_tickets)}, All open: {len(all_open)}")

asyncio.run(track_issues())

```

Core Concepts

Tickets

A **Ticket** is an issue tracking record:

- **subject** : Ticket title
- **description** : Detailed description
- **status** : Current status (OPEN, IN_PROGRESS, etc.)
- **priority** : Priority level (LOW, NORMAL, HIGH, CRITICAL)
- **assigned_to** : User assigned to the ticket

Status Workflow

Tickets flow through statuses:

- **OPEN**: New ticket, not yet started
- **IN_PROGRESS**: Actively being worked on
- **RESOLVED**: Solution implemented, awaiting verification
- **CLOSED**: Verified and closed

Priority Levels

Priority indicates urgency:

- **LOW**: Minor issue, no immediate impact
- **NORMAL**: Standard issue

- **HIGH**: Important, needs attention soon
- **CRITICAL**: Urgent, production impact

Views

View filters control which tickets you see:

- **ASSIGNED**: Tickets assigned to you
- **FOLLOWING**: Tickets you're following
- **ALL**: All tickets (requires permissions)

Ticket Management

Get Ticket by ID

```
# Get specific ticket
ticket = api.rootcause.get_ticket(12345)

if ticket:
    print(f"Ticket #{ticket.id}: {ticket.subject}")
    print(f"Status: {ticket.status}")
    print(f"Priority: {ticket.priority}")
    print(f"Assigned: {ticket.assigned_to}")
    print(f"Created: {ticket.created_date_time}")
    print(f"\nDescription:")
    print(ticket.description)
else:
    print("Ticket not found")
```

Get All Open Tickets

```
from pywats.domains.rootcause.models import TicketView

# Get all open tickets (view: all)
open_tickets = api.rootcause.get_open_tickets(view=TicketView.ALL)

print(f"== OPEN TICKETS ({len(open_tickets)}) ==")

for ticket in open_tickets:
    print(f"\n#{ticket.id}: {ticket.subject}")
    print(f" Priority: {ticket.priority}")
    print(f" Assigned: {ticket.assigned_to or 'Unassigned'}")
    print(f" Created: {ticket.created_date_time.strftime('%Y-%m-%d')}")
```

Get Active Tickets

```
# Get open + in-progress tickets
active = api.rootcause.get_active_tickets()

print(f"== ACTIVE TICKETS ({len(active)}) ==")

for ticket in active:
    status_icon = "🟡" if ticket.status == "OPEN" else "🔴"
    print(f"{status_icon} {ticket.id}: {ticket.subject}")
    print(f"  Status: {ticket.status}")
```

Status and Workflow

Query by Status

```
from pywats.domains.rootcause.models import TicketStatus

# Get tickets with specific status
open_tickets = api.rootcause.get_tickets(status=TicketStatus.OPEN)

print(f"Open tickets: {len(open_tickets)}")

# Get in-progress tickets
in_progress = api.rootcause.get_tickets(status=TicketStatus.IN_PROGRESS)

print(f"In progress: {len(in_progress)})")
```

Combine Multiple Statuses

```
# Get open OR in-progress tickets using "|" operator
active_status = f"{TicketStatus.OPEN}|{TicketStatus.IN_PROGRESS}"

active_tickets = api.rootcause.get_tickets(status=active_status)

print(f"Active tickets: {len(active_tickets)})")

# Breakdown by status
by_status = {}
for ticket in active_tickets:
    status = ticket.status
    by_status[status] = by_status.get(status, 0) + 1

for status, count in by_status.items():
    print(f"  {status}: {count}")
```

Track Status Changes

```
def track_ticket_workflow(ticket_id):
    """Show ticket history and status changes"""

    ticket = api.rootcause.get_ticket(ticket_id)

    if not ticket:
        print("Ticket not found")
        return

    print(f"==> TICKET #{ticket_id}: {ticket.subject} ===")
    print(f"Current Status: {ticket.status}")
    print(f"Created: {ticket.created_date_time}")

    # If ticket has updates/history
    if hasattr(ticket, 'updates') and ticket.updates:
        print("\nHistory:")
        for update in ticket.updates:
            print(f"  {update.date_time}: {update.description}")

    # Use it
track_ticket_workflow(12345)
```

Priority Management

Filter by Priority

```
from pywats.domains.rootcause.models import TicketPriority, TicketView

# Get high priority tickets
high_priority = api.rootcause.get_tickets(
    priority=TicketPriority.HIGH,
    view=TicketView.ALL
)

print(f"==> HIGH PRIORITY TICKETS ({len(high_priority)}) ===")

for ticket in high_priority:
    print(f"# {ticket.id}: {ticket.subject}")
    print(f"  Status: {ticket.status}")
    print(f"  Assigned: {ticket.assigned_to}")
```

Priority Report

```
from pywats.domains.rootcause.models import TicketView

def priority_report():
    """Generate report of tickets by priority"""

    # Get all active tickets
    active = api.rootcause.get_active_tickets(view=TicketView.ALL)

    # Count by priority
    by_priority = {}
    for ticket in active:
        priority = ticket.priority
        by_priority[priority] = by_priority.get(priority, 0) + 1

    print("=" * 60)
    print("ACTIVE TICKETS BY PRIORITY")
    print("=" * 60)

    # Define priority order
    priority_order = ["CRITICAL", "HIGH", "NORMAL", "LOW"]

    total = 0
    for priority in priority_order:
        count = by_priority.get(priority, 0)
        total += count

        if count > 0:
            icon = "\u2b57" if priority == "CRITICAL" else "\u2b58" if priority == "HIGH" else "\u2b59" if priority == "NORMAL" else "\u2b5c"
            print(f"{icon} {priority}:<12} {count:>4}")

    print("-" * 60)
    print(f"{'TOTAL':<14} {total:>4}")
    print("=" * 60)

# Use it
priority_report()
```

View Filtering

Assigned to Me

```
from pywats.domains.rootcause.models import TicketView

# Get tickets assigned to me
my_tickets = api.rootcause.get_active_tickets(view=TicketView.ASSIGNED)

print(f"==> MY TICKETS ({len(my_tickets)}) ==>")

for ticket in my_tickets:
    print(f"# {ticket.id}: {ticket.subject}")
    print(f"  Priority: {ticket.priority}")
    print(f"  Status: {ticket.status}")
```

Following

```
# Get tickets I'm following
following = api.rootcause.get_active_tickets(view=TicketView.FOLLOWING)

print(f"==> FOLLOWING ({len(following)}) ==>")

for ticket in following:
    print(f"# {ticket.id}: {ticket.subject}")
```

All Tickets (with permissions)

```
# Get all tickets (requires permissions)
all_tickets = api.rootcause.get_tickets(view=TicketView.ALL)

print(f"Total tickets: {len(all_tickets)}")

# Group by assigned user
by_user = {}
for ticket in all_tickets:
    user = ticket.assigned_to or "Unassigned"
    if user not in by_user:
        by_user[user] = []
    by_user[user].append(ticket)

print("\n==> TICKETS BY ASSIGNEE ==>")
for user, tickets in sorted(by_user.items()):
    print(f"{user}: {len(tickets)} tickets")
```

Search and Query

Search by Subject/Tags

```
# Search for tickets containing "calibration"
search_results = api.rootcause.get_tickets(search_string="calibration")

print(f"==> SEARCH: 'calibration' ({len(search_results)}) ==>")

for ticket in search_results:
    print(f"\n#{ticket.id}: {ticket.subject}")
    print(f"  Tags: {', '.join(ticket.tags) if ticket.tags else 'None'}")
```

Combined Search

```
from pywats.domains.rootcause.models import TicketStatus, TicketPriority

# Search for high priority open tickets about "ICT"
results = api.rootcause.get_tickets(
    status=TicketStatus.OPEN,
    priority=TicketPriority.HIGH,
    search_string="ICT"
)

print(f"==> HIGH PRIORITY OPEN ICT ISSUES ({len(results)}) ==>"

for ticket in results:
    print(f"\n#{ticket.id}: {ticket.subject}")
```

Find Tickets for Part Number

```
def find_tickets_for_part(part_number):
    """Find all tickets related to a part number"""

    # Search in subject and tags
    results = api.rootcause.get_tickets(search_string=part_number)

    print(f"==> TICKETS FOR {part_number} ({len(results)}) ==>"

    for ticket in results:
        print(f"\n#{ticket.id}: {ticket.subject}")
        print(f"  Status: {ticket.status}")
        print(f"  Priority: {ticket.priority}")
        print(f"  Created: {ticket.created_date_time.strftime('%Y-%m-%d')}")

    # Use it
    find_tickets_for_part("WIDGET-001")
```

Known Issues & Workarounds

Server Does Not Return Assignee Field

Problem: The WATS server does NOT return the `assignee` field in ticket API responses. After fetching a ticket via `get_ticket()`, the `assignee` field will always be `None`, even if the ticket is actually assigned to someone.

This causes issues because WATS enforces the business rule:

"Unassigned tickets must have status 'new'"

If you fetch a ticket and try to update it (e.g., change status to SOLVED), the update will fail with a 400 error because the server sees no assignee in your request.

Workaround: Always preserve and pass the assignee explicitly when modifying tickets.

```
# ❌ BAD - This will fail for assigned tickets with non-new status
ticket = api.rootcause.get_ticket(ticket_id)
ticket.status = TicketStatus.SOLVED
api.rootcause.update_ticket(ticket) # 400 Error: "Unassigned tickets must have status new"

# ✅ GOOD - Use change_status() with explicit assignee
api.rootcause.change_status(
    ticket_id=ticket_id,
    status=TicketStatus.SOLVED,
    assignee="user@example.com" # Must provide the current assignee!
)

# ✅ GOOD - Use add_comment() with explicit assignee
api.rootcause.add_comment(
    ticket_id=ticket_id,
    comment="This is a comment",
    assignee="user@example.com" # Must provide the current assignee!
)
```

Best Practices:

1. When creating workflows, store the assignee from `create_ticket()` or `assign_ticket()` results
2. Pass the assignee to all subsequent `add_comment()` and `change_status()` calls
3. Don't rely on `get_ticket()` to tell you who a ticket is assigned to
4. If using fixtures in tests, include the assignee in the fixture data

```
# Example: Store assignee when creating a ticket
ticket = api.rootcause.create_ticket(
    subject="Issue investigation",
    priority=TicketPriority.HIGH,
    assignee="user@example.com"
)

# The service preserves assignee in the result
stored_assignee = ticket.assignee # "user@example.com"

# Later, when updating the ticket, pass the stored assignee
api.rootcause.change_status(
    ticket_id=ticket.ticket_id,
    status=TicketStatus.SOLVED,
    assignee=stored_assignee
)
```

Advanced Usage

Ticket Dashboard

```
from pywats.domains.rootcause.models import TicketView

def ticket_dashboard():
    """Generate comprehensive ticket dashboard"""

    # Get all tickets
    all_tickets = api.rootcause.get_tickets(view=TicketView.ALL)

    print("=" * 70)
    print("TICKET DASHBOARD")
    print("=" * 70)

    # Summary by status
    by_status = {}
    for ticket in all_tickets:
        status = ticket.status
        by_status[status] = by_status.get(status, 0) + 1

    print("\nBy Status:")
    for status, count in sorted(by_status.items()):
        print(f" {status}: {count}")

    # Get active tickets
    active = [t for t in all_tickets if t.status in ["OPEN", "IN_PROGRESS"]]

    # Active by priority
    print(f"\nActive Tickets by Priority ({len(active)} total):")
    by_priority = {}
    for ticket in active:
        priority = ticket.priority
        by_priority[priority] = by_priority.get(priority, 0) + 1

    for priority in ["CRITICAL", "HIGH", "NORMAL", "LOW"]:
        count = by_priority.get(priority, 0)
        if count > 0:
            print(f" {priority}: {count}")

    # Unassigned
    unassigned = [t for t in active if not t.assigned_to]
    if unassigned:
        print(f"\n⚠ {len(unassigned)} unassigned active tickets")

    # Old tickets
    from datetime import datetime, timedelta
    old_cutoff = datetime.now() - timedelta(days=30)
    old_active = [t for t in active if t.created_date_time < old_cutoff]

    if old_active:
        print(f"\n⚠ {len(old_active)} active tickets older than 30 days")
```

```
print("=" * 70)
```

```
# Use it
```

```
ticket_dashboard()
```

Defect Tracking Integration

```
def track_defect_from_failure(uut_report_id):
    """Create/update ticket when a unit fails"""

    # Get the failed UUT report (from Report domain)
    uut_report = api.report.get_uut_report(uut_report_id)

    if not uut_report or uut_report.status != "Failed":
        print("UUT did not fail")
        return

    # Check if ticket already exists
    search_string = f"UUT:{uut_report.serial_number}"
    existing = api.rootcause.get_tickets(search_string=search_string)

    if existing:
        ticket = existing[0]
        print(f"Existing ticket: #{ticket.id}")
    else:
        # Create new ticket (implementation depends on API)
        print(f"Need to create ticket for {uut_report.serial_number}")

        subject = f"Failure: {uut_report.part_number} - {uut_report.serial_number}"
        description = f"Unit failed at {uut_report.completed_date_time}\n"
        description += f"Station: {uut_report.station}\n"
        description += f"Operator: {uut_report.operator}\n"

        # Add failed steps
        failed_steps = [s for s in uut_report.steps if s.status == "Failed"]
        description += f"\nFailed Steps:\n"
        for step in failed_steps:
            description += f"- {step.step_name}\n"

    # Use it
    track_defect_from_failure(12345)
```

Aging Report

```
from datetime import datetime, timedelta

def aging_report(days_threshold=30):
    """Find old active tickets"""

    cutoff = datetime.now() - timedelta(days=days_threshold)

    active = api.rootcause.get_active_tickets()

    old_tickets = [
        t for t in active
        if t.created_date_time < cutoff
    ]

    # Sort by age (oldest first)
    old_tickets.sort(key=lambda t: t.created_date_time)

    print(f"==== TICKETS OLDER THAN {days_threshold} DAYS ({len(old_tickets)}) ====")

    for ticket in old_tickets:
        age_days = (datetime.now() - ticket.created_date_time).days

        print(f"\n#{ticket.id}: {ticket.subject}")
        print(f"  Age: {age_days} days")
        print(f"  Status: {ticket.status}")
        print(f"  Priority: {ticket.priority}")
        print(f"  Assigned: {ticket.assigned_to or 'Unassigned'}")

    # Use it
    aging_report(days_threshold=60)
```

API Reference

RootCauseService Methods

Ticket Queries

- `get_ticket(ticket_id) → Optional[Ticket]` - Get specific ticket
- `get_tickets(status=None, view=None, search_string=None) → List[Ticket]` - Query tickets
- `get_open_tickets(view=None) → List[Ticket]` - Get open tickets
- `get_active_tickets(view=None) → List[Ticket]` - Get open + in-progress tickets

Models

Ticket

- `id` : int - Ticket ID
- `subject` : str - Ticket title
- `description` : str - Detailed description
- `status` : str - Status (OPEN, IN_PROGRESS, RESOLVED, CLOSED)
- `priority` : str - Priority (LOW, NORMAL, HIGH, CRITICAL)
- `assigned_to` : str - Assigned user
- `created_date_time` : datetime - Creation timestamp
- `tags` : List[str] - Tags for organization
- `updates` : List[TicketUpdate] - Update history

TicketStatus (Enum)

- OPEN - New ticket
- IN_PROGRESS - Being worked on
- RESOLVED - Solution implemented
- CLOSED - Verified and closed

TicketPriority (Enum)

- LOW - Minor issue
- NORMAL - Standard priority
- HIGH - Important
- CRITICAL - Urgent

TicketView (Enum)

- ASSIGNED - Tickets assigned to you
- FOLLOWING - Tickets you're following
- ALL - All tickets (requires permissions)

Best Practices

1. **Use appropriate priority** - Reserve CRITICAL for production impact
2. **Assign tickets promptly** - Avoid unassigned backlog
3. **Search before creating** - Avoid duplicate tickets

- 4. Update status regularly** - Keep workflow current
 - 5. Use tags** - Tag with part numbers, stations, categories
 - 6. Monitor aging tickets** - Review old active tickets regularly
 - 7. Link to evidence** - Reference UUT reports, measurements
 - 8. Close resolved tickets** - Verify and close when complete
-

See Also

- Report Domain - Link tickets to failed test reports
 - Production Domain - Track units with defects
 - Analytics Domain - Analyze failure trends
-

Source: docs/modules/software.md

Software Domain

The Software domain manages software packages and their distribution. Use this to track software versions, manage releases, organize packages with tags and folders, and associate files with packages. It supports versioning, status management (Development, Released, Obsolete), and tag-based organization.

Table of Contents

- Quick Start
 - Core Concepts
 - Package Management
 - Package Files
 - Tags and Organization
 - Virtual Folders
 - Advanced Usage
 - API Reference
-

Quick Start

Synchronous Usage

```
from pywats import pyWATS

# Initialize
api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your-api-token"
)

# Get all released packages
from pywats.domains.software.models import PackageStatus

released = api.software.get_packages(status=PackageStatus.RELEASED)

for pkg in released:
    print(f"{pkg.name} v{pkg.version} - {pkg.description}")

# Get specific package by name
package = api.software.get_package_by_name("FirmwareUpdate")

if package:
    print(f"Package: {package.name}")
    print(f"Version: {package.version}")
    print(f"Status: {package.status}")
    print(f"Files: {len(package.files)}")

    # List package files
    for file in package.files:
        print(f" - {file.file_name} ({file.file_size} bytes)")

# Get latest released version
latest = api.software.get_released_package("FirmwareUpdate")
```

Asynchronous Usage

For concurrent requests and better performance:

```

import asyncio
from pywats import AsyncWATS
from pywats.domains.software.models import PackageStatus

async def manage_software():
    async with AsyncWATS(
        base_url="https://your-wats-server.com",
        token="your-api-token"
    ) as api:
        # Fetch multiple package statuses concurrently
        released, development = await asyncio.gather(
            api.software.get_packages(status=PackageStatus.RELEASED),
            api.software.get_packages(status=PackageStatus.DEVELOPMENT)
        )

        print(f"Released: {len(released)}, In Development: {len(development)}")

asyncio.run(manage_software())

```

Core Concepts

Packages

A **Package** is a versioned software distribution:

- name : Package name (unique)
- version : Version string
- status : DEVELOPMENT, RELEASED, or OBSOLETE
- description : Package description
- files : Associated files

Package Status

Packages can be in three states:

- **DEVELOPMENT**: Under development, not ready for release
- **RELEASED**: Released and ready for use
- **OBSOLETE**: Deprecated, no longer recommended

Tags

Tags are key-value pairs for organizing packages:

- tag : Tag name (e.g., "product", "type")
- value : Tag value (e.g., "WIDGET-001", "firmware")

Virtual Folders

Virtual Folders organize packages hierarchically:

- Folder-like structure without physical directories
 - Packages can belong to multiple folders
 - Used for logical organization (e.g., by product line, by type)
-

Package Management

List All Packages

```
# Get all packages
all_packages = api.software.get_packages()

print(f"Total packages: {len(all_packages)}")

for pkg in all_packages:
    print(f"{pkg.name} v{pkg.version} [{pkg.status}]")
```

Filter by Status

```
from pywats.domains.software.models import PackageStatus

# Get only released packages
released = api.software.get_packages(status=PackageStatus.RELEASED)

print("== RELEASED PACKAGES ==")
for pkg in released:
    print(f"{pkg.name} v{pkg.version}")

# Get development packages
dev_packages = api.software.get_packages(status=PackageStatus.DEVELOPMENT)

print(f"\n{len(dev_packages)} packages in development")
```

Get Package by ID

```
# Get specific package
package = api.software.get_package(12345)

if package:
    print(f"Package: {package.name}")
    print(f"Version: {package.version}")
    print(f"Status: {package.status}")
    print(f"Description: {package.description}")
    print(f"Created: {package.created_date_time}")
else:
    print("Package not found")
```

Get Package by Name

```
# Get latest version of a package by name
package = api.software.get_package_by_name("FirmwareUpdate")

if package:
    print(f"Found: {package.name} v{package.version}")
else:
    print("Package not found")

# Get specific version and status
package = api.software.get_package_by_name(
    "FirmwareUpdate",
    version="2.1.0",
    status=PackageStatus.RELEASED
)
```

Get Latest Released Package

```
# Get the latest released version
latest = api.software.get_released_package("FirmwareUpdate")

if latest:
    print(f"Latest released: v{latest.version}")
    print(f"Released: {latest.created_date_time}")
else:
    print("No released version found")
```

Package Files

List Package Files

```
# Get package and its files
package = api.software.get_package_by_name("FirmwareUpdate")

if package and package.files:
    print(f"== FILE IN {package.name} v{package.version} ==")

    for file in package.files:
        print(f"\n{file.file_name}:")
        print(f"  Size: {file.file_size:,} bytes")
        print(f"  Type: {file.file_type}")
        print(f"  MD5: {file.md5_checksum}")

        if file.description:
            print(f"  Description: {file.description}")
    else:
        print("No files found")
```

Download Package File

```
import os

def download_package_files(package_name, destination_folder):
    """Download all files from a package"""

    # Get package
    package = api.software.get_released_package(package_name)

    if not package:
        print(f"Package '{package_name}' not found")
        return

    if not package.files:
        print(f"No files in package '{package_name}'")
        return

    # Create destination folder
    os.makedirs(destination_folder, exist_ok=True)

    print(f"Downloading {len(package.files)} files from {package.name} v{package.version}...")

    for file in package.files:
        file_path = os.path.join(destination_folder, file.file_name)

        # Download using software service (method depends on implementation)
        # This is a placeholder - actual implementation may vary
        print(f" - {file.file_name} ({file.file_size:,} bytes)")

        # Verify checksum after download
        if hasattr(file, 'md5_checksum'):
            print(f"    MD5: {file.md5_checksum}")

    print(f"Downloaded to: {destination_folder}")

# Use it
download_package_files("FirmwareUpdate", "C:\\\\Downloads\\\\Firmware")
```

Tags and Organization

Query Packages by Tag

```
# Get packages with a specific tag
packages = api.software.get_packages_by_tag(
    tag="product",
    value="WIDGET-001"
)

print(f"== PACKAGE FOR WIDGET-001 ==")
for pkg in packages:
    print(f"{pkg.name} v{pkg.version}")

# Filter by status too
released_packages = api.software.get_packages_by_tag(
    tag="product",
    value="WIDGET-001",
    status=PackageStatus.RELEASED
)
```

List Package Tags

```
# Get package and inspect tags
package = api.software.get_package_by_name("FirmwareUpdate")

if package and package.tags:
    print(f"== TAGS FOR {package.name} ==")

    for tag in package.tags:
        print(f"{tag.tag}: {tag.value}")
else:
    print("No tags found")
```

Organize by Tags

```
def list_packages_by_product():
    """Group packages by product tag"""

    # Get all released packages
    packages = api.software.get_packages(status=PackageStatus.RELEASED)

    # Group by product tag
    by_product = {}

    for pkg in packages:
        if pkg.tags:
            for tag in pkg.tags:
                if tag.tag == "product":
                    product = tag.value
                    if product not in by_product:
                        by_product[product] = []
                    by_product[product].append(pkg)

    # Display
    for product, pkgs in by_product.items():
        print(f"\n==== {product} ====")
        for pkg in pkgs:
            print(f"  {pkg.name} v{pkg.version}")

# Use it
list_packages_by_product()
```

Virtual Folders

List Folders

```
# Get all virtual folders
folders = api.software.get_virtual_folders()

print("==== VIRTUAL FOLDERS ====")
for folder in folders:
    print(f"{folder.path}")
    print(f"  Packages: {folder.package_count}")
```

Get Packages in Folder

```
# Get packages in a specific folder
folder_packages = api.software.get_packages_in_folder("Firmware/Stable")

print(f"== PACKAGE FOLDERS ==")
for pkg in folder_packages:
    print(f"{pkg.name} v{pkg.version}")
```

Folder Hierarchy

```
def show_folder_tree():
    """Display virtual folder hierarchy"""

    folders = api.software.get_virtual_folders()

    # Sort by path
    folders.sort(key=lambda f: f.path)

    print("== PACKAGE FOLDERS ==")
    for folder in folders:
        # Calculate indent based on path depth
        depth = folder.path.count('/') + folder.path.count('\\')
        indent = " " * depth
        folder_name = folder.path.split('/')[-1]

        print(f"{indent}{folder_name}/ ({folder.package_count})")

    # Use it
    show_folder_tree()
```

Advanced Usage

Version Comparison

```
from packaging import version

def get_latest_version(package_name):
    """Find latest version across all statuses"""

    # Get all packages with this name
    packages = api.software.get_packages()

    # Filter by name
    matching = [p for p in packages if p.name == package_name]

    if not matching:
        print(f"No packages found with name '{package_name}'")
        return None

    # Sort by version
    matching.sort(key=lambda p: version.parse(p.version), reverse=True)

    latest = matching[0]

    print(f"== VERSIONS OF {package_name} ==")
    for pkg in matching:
        current = " <- LATEST" if pkg == latest else ""
        print(f"v{pkg.version} [{pkg.status}]{current}")

    return latest

# Use it
latest = get_latest_version("FirmwareUpdate")
```

Package Audit

```
from datetime import datetime, timedelta

def audit_packages(days=30):
    """Find packages that need attention"""

    cutoff = datetime.now() - timedelta(days=days)

    packages = api.software.get_packages()

    print(f"== PACKAGE AUDIT (last {days} days) ==\n")

    # Old development packages
    old_dev = [
        p for p in packages
        if p.status == PackageStatus.DEVELOPMENT
        and p.created_date_time < cutoff
    ]

    if old_dev:
        print(f"\u25b6 {len(old_dev)} development packages older than {days} days:")
        for pkg in old_dev:
            age = (datetime.now() - pkg.created_date_time).days
            print(f"  - {pkg.name} v{pkg.version} ({age} days old)")

    # Packages with no files
    no_files = [p for p in packages if not p.files]

    if no_files:
        print(f"\n\u25b6 {len(no_files)} packages with no files:")
        for pkg in no_files:
            print(f"  - {pkg.name} v{pkg.version} [{pkg.status}]")

    # Released packages
    released = [p for p in packages if p.status == PackageStatus.RELEASED]
    print(f"\n✓ {len(released)} released packages")

    # Use it
    audit_packages(days=90)
```

Package Distribution Report

```
def package_distribution_report():
    """Show package distribution by status"""

    packages = api.software.get_packages()

    # Count by status
    by_status = {}
    for pkg in packages:
        status = pkg.status.name if hasattr(pkg.status, 'name') else str(pkg.status)
        by_status[status] = by_status.get(status, 0) + 1

    # Count by tag
    by_tag = {}
    for pkg in packages:
        if pkg.tags:
            for tag in pkg.tags:
                key = f"{tag.tag}:{tag.value}"
                by_tag[key] = by_tag.get(key, 0) + 1

    # Display
    print("=" * 60)
    print("PACKAGE DISTRIBUTION REPORT")
    print("=" * 60)

    print(f"\nTotal Packages: {len(packages)}")

    print("\nBy Status:")
    for status, count in sorted(by_status.items()):
        print(f"  {status}: {count}")

    print("\nTop Tags:")
    sorted_tags = sorted(by_tag.items(), key=lambda x: x[1], reverse=True)
    for tag, count in sorted_tags[:10]:
        print(f"  {tag}: {count}")

    print("=" * 60)

    # Use it
    package_distribution_report()
```

API Reference

SoftwareService Methods

Package Queries

- `get_packages(status=None)` → `List[Package]` - Get all packages, optionally filtered by status

- `get_package(package_id) → Optional[Package]` - Get package by ID
- `get_package_by_name(name, status=None, version=None) → Optional[Package]` - Get package by name
- `get_released_package(name) → Optional[Package]` - Get latest released version

Tag Queries

- `get_packages_by_tag(tag, value, status=None) → List[Package]` - Get packages with specific tag

Virtual Folder Queries

- `get_virtual_folders() → List[VirtualFolder]` - Get all folders
- `get_packages_in_folder(folder_path) → List[Package]` - Get packages in folder

Models

Package

- `id : int` - Package ID
- `name : str` - Package name
- `version : str` - Version string
- `status : PackageStatus` - Status (DEVELOPMENT, RELEASED, OBSOLETE)
- `description : str` - Description
- `created_date_time : datetime` - Creation timestamp
- `files : List[PackageFile]` - Associated files
- `tags : List[PackageTag]` - Tags

PackageFile

- `id : int` - File ID
- `file_name : str` - File name
- `file_size : int` - File size in bytes
- `file_type : str` - MIME type
- `md5_checksum : str` - MD5 hash
- `description : str` - File description

PackageTag

- `tag : str` - Tag name
- `value : str` - Tag value

VirtualFolder

- `id : int` - Folder ID
- `path : str` - Folder path
- `package_count : int` - Number of packages

PackageStatus (Enum)

- `DEVELOPMENT` - Under development
 - `RELEASED` - Released and ready
 - `OBSOLETE` - Deprecated
-

Best Practices

1. **Use status filtering** - Filter by `RELEASED` for production deployments
 2. **Version semantically** - Use semantic versioning (`major.minor.patch`)
 3. **Tag consistently** - Establish tag naming conventions
 4. **Verify checksums** - Always verify MD5 checksums after download
 5. **Organize with folders** - Use virtual folders for logical grouping
 6. **Archive old versions** - Mark as `OBSOLETE` instead of deleting
 7. **Document packages** - Provide clear descriptions
 8. **Track associations** - Tag packages with product/component info
-

See Also

- Product Domain - Products and revisions
- Production Domain - Unit production and assembly
- Asset Domain - Equipment and calibration

SCIM - User Provisioning

The SCIM (System for Cross-domain Identity Management) domain provides API access for automatic user provisioning from Azure Active Directory to WATS.

Overview

SCIM is an industry standard protocol (RFC 7644) for managing user identities across cloud services. The WATS SCIM endpoint enables automatic user provisioning from Azure AD, supporting:

- User creation and deletion
- User activation/deactivation
- User attribute updates (display name, etc.)
- Token generation for Azure AD configuration

Quick Start

Synchronous Usage

```
from pywats import pyWATS

api = pyWATS(
    base_url="https://your-wats-server.com",
    token="your-api-token"
)

# Get provisioning token for Azure AD configuration
token = api.scim.get_token(duration_days=90)
if token:
    print(f"Configure Azure with: {token.token[:50]}...")

# List all SCIM users
users = api.scim.get_users()
for user in users.resources or []:
    status = "active" if user.active else "inactive"
    print(f"{user.user_name}: {user.display_name} ({status})")
```

Asynchronous Usage

For concurrent requests and better performance:

```

import asyncio
from pywats import AsyncWATS

async def provision_users():
    async with AsyncWATS(
        base_url="https://your-wats-server.com",
        token="your-api-token"
    ) as api:
        # Get token and users concurrently
        token, users = await asyncio.gather(
            api.scim.get_token(duration_days=90),
            api.scim.get_users()
        )

    print(f"Token valid until: {token.expires_utc if token else 'N/A'}")
    print(f"Total users: {users.total_results}")

asyncio.run(provision_users())

```

Service Methods

Token Generation

```

# Get JWT token for Azure AD provisioning configuration
token = api.scim.get_token(duration_days=90)
if token:
    print(f"Token expires: {token.expires_utc}")
    print(f"Token: {token.token}")

```

User Listing

```

# Get all SCIM users (single page)
response = api.scim.get_users()
print(f"Total users: {response.total_results}")

for user in response.resources or []:
    print(f" {user.id}: {user.user_name}")

```

Paginated User Iteration

```
# Iterate over ALL users efficiently (automatic pagination)
for user in api.scim.iter_users(page_size=100):
    print(f"{user.user_name}: {user.display_name}")

# With max limit
for user in api.scim.iter_users(page_size=50, max_users=200):
    process_user(user)

# With progress callback
def on_page(page_num, items_so_far, total):
    print(f"Fetched page {page_num}, {items_so_far}/{total} users")

for user in api.scim.iter_users(on_page=on_page):
    sync_user(user)
```

User Creation

```
from pywats import ScimUser, ScimUserName, ScimUserEmail

# Create a new user
user = ScimUser(
    user_name="john.doe@example.com",
    display_name="John Doe",
    active=True,
    name=ScimUserName(given_name="John", family_name="Doe"),
    emails=[ScimUserEmail(value="john.doe@example.com", type="work", primary=True)]
)

created = api.scim.create_user(user)
if created:
    print(f"Created user ID: {created.id}")
```

User Retrieval

```
# Get user by ID
user = api.scim.get_user("a1b2c3d4-e5f6-7890-abcd-ef1234567890")
if user:
    print(f"User: {user.display_name}")

# Get user by username
user = api.scim.get_user_by_username("john.doe@example.com")
if user:
    print(f"User ID: {user.id}")
```

User Updates

```
from pywats import ScimPatchRequest, ScimPatchOperation

# Update display name (convenience method)
updated = api.scim.update_display_name("user-id", "John Smith")

# Deactivate a user (convenience method)
deactivated = api.scim.deactivate_user("user-id")

# Set active status (convenience method)
activated = api.scim.set_user_active("user-id", active=True)

# Manual patch operation
patch = ScimPatchRequest(
    operations=[
        ScimPatchOperation(op="replace", path="displayName", value="Jane Doe"),
        ScimPatchOperation(op="replace", path="active", value=False)
    ]
)
updated = api.scim.update_user("user-id", patch)
```

User Deletion

```
# Delete user by ID
api.scim.delete_user("a1b2c3d4-e5f6-7890-abcd-ef1234567890")
```

Models

ScimUser

Main user resource model.

Field	Type	Description
id	str	Unique user identifier (GUID)
user_name	str	Username (typically email address)
display_name	str	Display name
active	bool	Whether user is active
external_id	str	External ID from Azure AD
name	ScimUserName	Structured name components
emails	List[ScimUserEmail]	Email addresses
schemas	List[str]	SCIM schemas
meta	Dict	SCIM metadata

ScimUserName

User name components.

Field	Type	Description
formatted	str	Full formatted name
given_name	str	First name
family_name	str	Last name (surname)

ScimUserEmail

User email entry.

Field	Type	Description
value	str	Email address
type	str	Email type (work, home, etc.)
primary	bool	Whether primary email

ScimToken

JWT token response.

Field	Type	Description
token	str	JWT token string
expires_utc	datetime	Token expiration (UTC)
duration_days	int	Token validity in days

ScimPatchRequest

SCIM patch request body.

Field	Type	Description
schemas	List[str]	Must include patch schema
operations	List[ScimPatchOperation]	Patch operations

ScimPatchOperation

Single patch operation.

Field	Type	Description
op	str	Operation type (only "replace" supported)
path	str	Attribute path
value	Any	New value

ScimListResponse

Paginated list response.

Field	Type	Description
total_results	int	Total result count
items_per_page	int	Items per page
start_index	int	Start index
resources	List[ScimUser]	User resources
schemas	List[str]	Response schemas

API Endpoints

Method	Endpoint	Service Method
GET	/api/SCIM/v2/Token	get_token()
GET	/api/SCIM/v2/Users	get_users() , iter_users()
POST	/api/SCIM/v2/Users	create_user()
GET	/api/SCIM/v2/Users/{id}	get_user()
DELETE	/api/SCIM/v2/Users/{id}	delete_user()
PATCH	/api/SCIM/v2/Users/{id}	update_user()
GET	/api/SCIM/v2/Users/username={userName}	get_user_by_username()

Azure AD Configuration

To configure Azure AD for automatic provisioning:

1. Generate a provisioning token:

```
python token = api.scim.get_token(duration_days=90)
```

2. In Azure AD Enterprise Applications:

3. Select your WATS application

4. Go to Provisioning

5. Set Provisioning Mode to "Automatic"

6. Enter your WATS SCIM endpoint URL: <https://your-wats-server.com/api/SCIM/v2>

7. Use the generated token as the "Secret Token"

8. Configure attribute mappings as needed

9. Enable provisioning

Error Handling

```
from pywats import PyWATSError, NotFoundError

try:
    user = api.scim.get_user("non-existent-id")
except NotFoundError:
    print("User not found")
except PyWATSError as e:
    print(f"SCIM error: {e}")
```

Notes

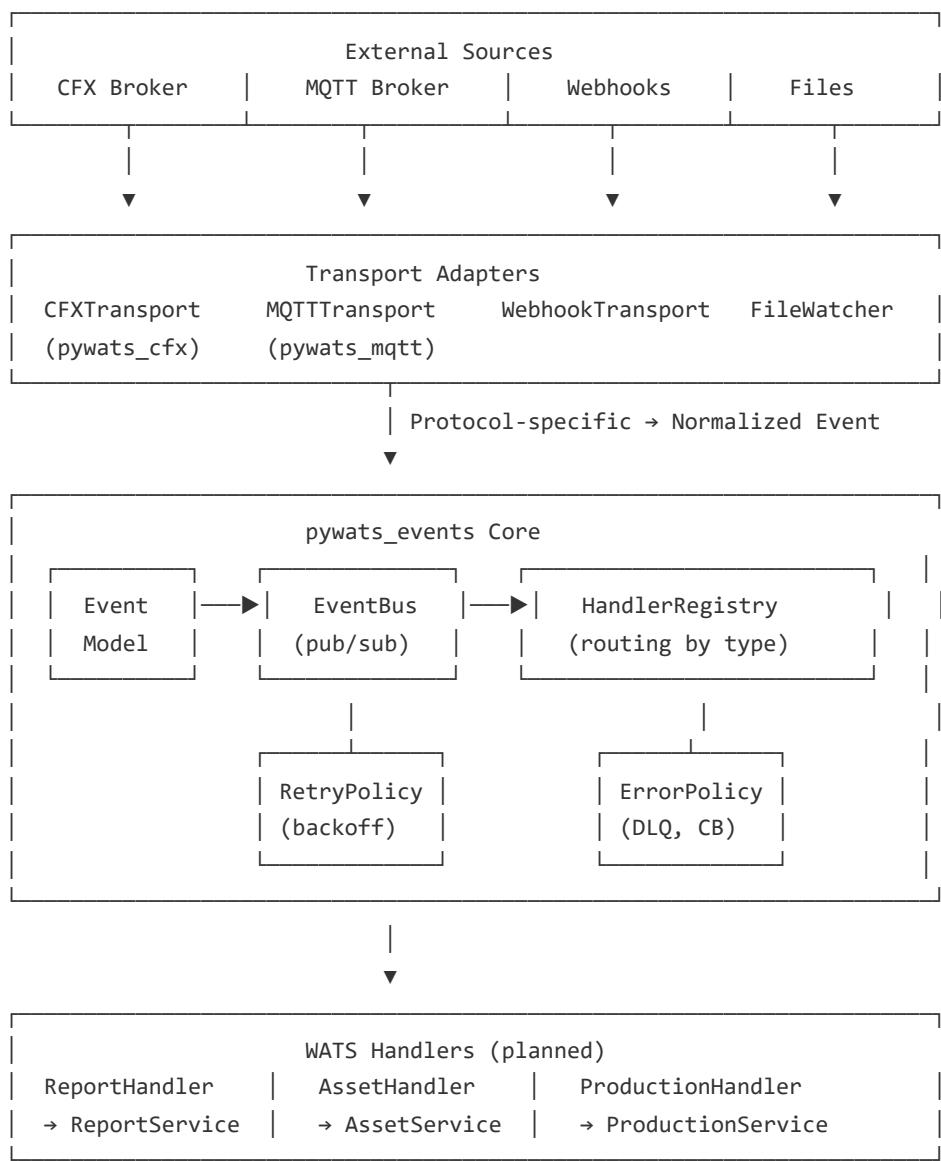
- SCIM PATCH operations only support "replace" operation type
- Users provisioned via SCIM should be managed through Azure AD
- Token duration default is 90 days if not specified
- All models use snake_case field names (camelCase aliases handled automatically)

Source: [docs/modules/EVENTS.md](#)

Event Architecture

The pyWATS event system provides a protocol-agnostic way to integrate with external message sources (IPC-CFX, MQTT, webhooks) and route events to appropriate handlers.

Architecture Overview



Packages

pywats_events (Core)

The protocol-agnostic event system foundation:

```

from pywats_events import Event, EventType, EventBus, BaseHandler

# Create an event
event = Event(
    event_type=EventType.TEST_RESULT,
    payload={"unit_id": "SN-123", "result": "pass"},
    source="test-station"
)

# Create a handler
class MyHandler(BaseHandler):
    @property
    def event_types(self):
        return [EventType.TEST_RESULT]

    async def handle(self, event):
        print(f"Processing: {event.payload['unit_id']}")

# Wire it up
bus = EventBus()
bus.subscribe(MyHandler())
bus.start()
bus.publish(event)

```

pywats_cfx (IPC-CFX Adapter)

Adapter for IPC-CFX factory messaging standard:

```

from pywats_cfx import CFXTransport, CFXConfig
from pywats_events import EventBus

# Configure CFX connection
config = CFXConfig(
    amqp=AMQPConfig(host="cfx-broker.factory.local"),
    endpoint=EndpointConfig(cfx_handle="//Company/WATS/Station1")
)

# Create transport and connect to event bus
transport = CFXTransport(config)
bus = EventBus()
bus.register_transport(transport)

# CFX messages automatically converted to domain events
bus.start() # Starts receiving CFX messages

```

Event Types

Events are categorized by domain:

Category	Event Types	Description
Test	TEST_RESULT , TEST_STARTED , INSPECTION_RESULT	Test and inspection results
Asset	ASSET_FAULT , ASSET_STATE_CHANGED , ASSET_MAINTENANCE	Equipment status and faults
Material	MATERIAL_INSTALLED , MATERIAL_CONSUMED	BOM and component tracking
Production	WORK_STARTED , WORK_COMPLETED , UNIT_DISQUALIFIED	Production flow events
System	TRANSPORT_CONNECTED , HANDLER_ERROR , EVENT_DEAD_LETTER	Internal system events

CFX Message Mapping

IPC-CFX messages are converted to normalized domain events:

CFX Message	Domain Event	WATS Handler Action
UnitsTested	TestResultEvent	ReportService.submit()
UnitsInspected	TestResultEvent	ReportService.submit()
MaterialsInstalled	MaterialInstalledEvent	Component traceability
FaultOccurred	AssetFaultEvent	AssetService fault logging
StationStateChanged	AssetStateChangedEvent	Equipment status
WorkStarted/Completed	WorkStartedEvent/WorkCompletedEvent	Production tracking

CFX Sample Explorer

Use the explorer CLI to understand CFX message formats:

```
# List all available samples
python -m pywats_cfx.explorer --list

# View raw CFX message
python -m pywats_cfx.explorer --sample units_tested_ict

# Convert to domain event with WATS mapping hints
python -m pywats_cfx.explorer --convert units_tested_ict
```

Available samples:

- **Test:** units_tested_ict , units_tested_fct , units_inspected_aoi , units_inspected_spi
- **Production:** work_started , work_completed_pass , work_completed_fail
- **Materials:** materials_smt , materials_th , materials_loaded
- **Faults:** fault_temp , fault_feeder , fault_cleared

Policies

Retry Policy

Configure retry behavior for failed handlers:

```
from pywats_events.policies import RetryPolicy

policy = RetryPolicy(
    max_retries=3,
    initial_delay=1.0,      # 1 second
    max_delay=60.0,         # Max 60 seconds
    exponential_base=2.0,   # Exponential backoff
    jitter=True             # Add randomness
)

# Filter which exceptions to retry
policy.retry_on(ConnectionError, TimeoutError)
policy.no_retry_on(ValidationError) # Never retry these
```

Error Policy

Handle permanently failed events:

```

from pywats_events.policies import ErrorPolicy, DeadLetterQueue, CircuitBreaker

dlq = DeadLetterQueue(max_size=1000)
cb = CircuitBreaker(
    failure_threshold=5,      # Open after 5 failures
    reset_timeout=30.0        # Try again after 30 seconds
)

policy = ErrorPolicy(dead_letter_queue=dlq, circuit_breaker=cb)

# Get failed events for analysis/replay
for entry in dlq.get_entries():
    print(f"Failed: {entry.event.id} - {entry.error}")

```

Testing

Use `MockTransport` for testing without external dependencies:

```

from pywats_events import EventBus, Event, EventType
from pywats_events.transports import MockTransport

def test_my_handler():
    bus = EventBus()
    transport = MockTransport()
    bus.register_transport(transport)

    received = []
    class TestHandler(BaseHandler):
        def event_types(self): return [EventType.TEST_RESULT]
        def handle(self, event): received.append(event)

    bus.subscribe(TestHandler())
    bus.start()

    # Inject test event
    transport.inject_event(Event(
        event_type=EventType.TEST_RESULT,
        payload={"unit_id": "TEST-123", "passed": True}
    ))

    assert len(received) == 1

```

Next Steps

The WATS handlers that consume domain events and call pyWATS services are planned:

```
# Future: WATS handlers (not yet implemented)
class ReportHandler(BaseHandler):
    def __init__(self, report_service: ReportService):
        self.report_service = report_service

    @property
    def event_types(self):
        return [EventType.TEST_RESULT]

    async def handle(self, event: Event):
        result = TestResultEvent(**event.payload)
        report = self._build_uut_report(result)
        await self.report_service.submit(report)
```

See the CFX explorer output for mapping hints when implementing handlers.

Usage Guides

Source: [docs/usage/report-module.md](#)

Report Module Usage Guide

Overview

The Report module handles test report submission and querying. It supports two report types:

- **UUT Reports** (Unit Under Test) - Test results for passing/failing units
- **UUR Reports** (Unit Under Repair) - Repair/rework documentation

Quick Start

```
from pywats import pyWATS
from pywats.models import UUTReport
from pywats.tools.test_uut import TestUUT

api = pyWATS(base_url="https://wats.example.com", token="credentials")

# Create report using factory
uut = TestUUT(
    part_number="PART-001",
    serial_number="SN-12345",
    revision="A",
    operator="John Doe",
    purpose=10 # Test
)

# Add test steps
root = uut.get_root()
root.add_numeric_step(
    name="Voltage Test",
    value=5.0,
    unit="V",
    comp_op="GELE",
    low_limit=4.5,
    high_limit=5.5,
    status="Passed"
)

# Submit report
report = uut.to_report()
api.report.send_uut_report(report)
```

Factory Methods (RECOMMENDED)

Why Use Factory Methods?

The `TestUUT` factory provides:

- Automatic report structure creation
- Proper sequence hierarchy
- Convenient step creation methods
- Automatic timestamp management
- Validation during construction

Creating a Report with TestUUT

```
from pywats.tools.test_uut import TestUUT

# Initialize with basic info
uut = TestUUT(
    part_number="MODULE-100",
    serial_number="MOD-2025-001",
    revision="B",
    operator="Test Operator",
    station="Station-5",
    purpose=10, # Test purpose
    operation_type=100 # Operation/process code (optional)
)

# Get root sequence to add steps
root = uut.get_root()

# Add test steps (see step types below)
root.add_numeric_step(...)
root.add_boolean_step(...)

# Convert to report and submit
report = uut.to_report()
api.report.send_uut_report(report)
```

Factory Constructor Parameters

```
TestUUT(
    part_number: str,                      # Product part number (required)
    serial_number: str,                     # Unit serial number (required)
    revision: str = "A",                   # Product revision
    operator: str = "Unknown",             # Operator name
    station: str = "",                    # Test station ID
    purpose: int = 10,                    # Purpose code (10=Test, 20=Debug, etc.)
    operation_type: Optional[int] = None,  # Process/operation code
    batch_serial_number: str = "",        # Batch/lot number
    start_time: Optional[datetime] = None # Auto-set if not provided
)
```

Step Types and Methods

All step methods are called on a sequence (typically `root = uut.get_root()`).

1. Numeric Step (Single Measurement)

```
root.add_numeric_step(
    name="Voltage Output",
    value=5.02,
    unit="V",
    comp_op="GELE",      # Comparison: GELE, GT, LT, EQ, NE, GE, LE, LOG
    low_limit=4.5,
    high_limit=5.5,
    status="Passed"      # "Passed", "Failed", "Done", "Skipped", etc.
)
```

Comparison Operators:

- GELE - Greater or Equal, Less or Equal (in range)
- GT - Greater Than
- LT - Less Than
- GE - Greater or Equal
- LE - Less or Equal
- EQ - Equal
- NE - Not Equal
- LOG - Logarithmic

2. Multi-Numeric Step (Multiple Measurements)

```
root.add_multi_numeric_step(
    name="Power Rails",
    measurements=[
        {
            "name": "3.3V Rail",
            "value": 3.31,
            "unit": "V",
            "comp_op": "GELE",
            "low_limit": 3.0,
            "high_limit": 3.6,
            "status": "Passed"
        },
        {
            "name": "5V Rail",
            "value": 5.05,
            "unit": "V",
            "comp_op": "GELE",
            "low_limit": 4.75,
            "high_limit": 5.25,
            "status": "Passed"
        }
    ],
    status="Passed"  # Overall step status
)
```

3. Boolean Step (Pass/Fail Test)

```
root.add_boolean_step(  
    name="LED Test",  
    value=True,           # True = Passed, False = Failed  
    status="Passed"  
)
```

4. Multi-Boolean Step (Multiple Pass/Fail Tests)

```
root.add_multi_boolean_step(  
    name="Digital I/O Test",  
    measurements=[  
        {"name": "Output 1", "value": True, "status": "Passed"},  
        {"name": "Output 2", "value": True, "status": "Passed"},  
        {"name": "Output 3", "value": False, "status": "Failed"}  
    status="Failed"  # Overall - failed if any measurement failed  
)
```

5. String Step (Text Value)

```
root.add_string_step(  
    name="Firmware Version",  
    value="v2.1.5",  
    status="Done"  
)
```

6. Multi-String Step (Multiple Text Values)

```
root.add_multi_string_step(  
    name="Component Versions",  
    measurements=[  
        {"name": "Bootloader", "value": "v1.2.0", "status": "Done"},  
        {"name": "Application", "value": "v2.1.5", "status": "Done"},  
        {"name": "FPGA", "value": "v3.0.1", "status": "Done"}  
    status="Done"  
)
```

7. Nested Sequences (Test Groups)

```
# Create a sub-sequence for grouping related tests
power_tests = root.add_sequence("Power Supply Tests")

# Add steps to the sub-sequence
power_tests.add_numeric_step(...)
power_tests.add_numeric_step(...)

# Create another sub-sequence
io_tests = root.add_sequence("I/O Tests")
io_tests.add_boolean_step(...)
```

8. Chart Step (Graphs/Plots)

```
root.add_chart_step(
    name="Frequency Response",
    chart_type="LINE", # LINE, LINE_LOG_X, LINE_LOG_Y, LINE_LOG_XY
    series=[
        {
            "name": "Magnitude",
            "x_values": [100, 1000, 10000, 100000],
            "y_values": [-0.5, -0.1, -3.0, -12.0],
            "x_unit": "Hz",
            "y_unit": "dB"
        }
    ],
    status="Done"
)
```

Complete Example: Complex Test Report

```
from pywats.tools.test_uut import TestUUT
from datetime import datetime

# Create test report
uut = TestUUT(
    part_number="POWER-MODULE-500W",
    serial_number="PM500-2025-12345",
    revision="C",
    operator="Jane Smith",
    station="Final-Test-3",
    purpose=10,
    operation_type=50, # Final test
    batch_serial_number="BATCH-2025-W50",
    start_time=datetime.now()
)

root = uut.get_root()

# 1. Initialization tests
init_seq = root.add_sequence("Initialization")
init_seq.add_string_step("Firmware Version", "v3.2.1", "Done")
init_seq.add_boolean_step("Self-Test", True, "Passed")

# 2. Power output tests
power_seq = root.add_sequence("Power Output Tests")
power_seq.add_multi_numeric_step(
    name="Output Voltages",
    measurements=[
        {
            "name": "12V Output",
            "value": 12.05,
            "unit": "V",
            "comp_op": "GELE",
            "low_limit": 11.4,
            "high_limit": 12.6,
            "status": "Passed"
        },
        {
            "name": "5V Output",
            "value": 5.02,
            "unit": "V",
            "comp_op": "GELE",
            "low_limit": 4.75,
            "high_limit": 5.25,
            "status": "Passed"
        },
        {
            "name": "3.3V Output",
            "value": 3.31,
            "unit": "V",
            "comp_op": "GELE",
            "low_limit": 3.14,
            "status": "Passed"
        }
    ]
)
```

```

        "high_limit": 3.47,
        "status": "Passed"
    }
],
status="Passed"
)

# 3. Load regulation test
load_seq = root.add_sequence("Load Regulation")
load_seq.add_numeric_step(
    name="12V @ 10A",
    value=11.98,
    unit="V",
    comp_op="GELE",
    low_limit=11.4,
    high_limit=12.6,
    status="Passed"
)
load_seq.add_numeric_step(
    name="12V @ 40A",
    value=11.92,
    unit="V",
    comp_op="GELE",
    low_limit=11.4,
    high_limit=12.6,
    status="Passed"
)
load_seq.add_numeric_step(
    name="12V @ 10A",
    value=11.98,
    unit="V",
    comp_op="GELE",
    low_limit=11.4,
    high_limit=12.6,
    status="Passed"
)

# 4. Protection tests
protection_seq = root.add_sequence("Protection Tests")
protection_seq.add_multi_boolean_step(
    name="Protection Circuits",
    measurements=[
        {"name": "Over-Voltage", "value": True, "status": "Passed"},
        {"name": "Over-Current", "value": True, "status": "Passed"},
        {"name": "Over-Temperature", "value": True, "status": "Passed"},
        {"name": "Short-Circuit", "value": True, "status": "Passed"}
    ],
    status="Passed"
)

# 5. Efficiency test with chart
efficiency_seq = root.add_sequence("Efficiency Test")
efficiency_seq.add_chart_step(
    name="Efficiency vs Load",
    chart_type="LINE",
    series:[
        {
            "name": "Efficiency",
            "x_values": [10, 20, 30, 40], # Load in amps
            "y_values": [88.5, 91.2, 92.1, 91.8], # Efficiency %
            "x_unit": "A",
            "y_unit": "%"
        }
    ],
    status="Done"
)

```

```
)  
  
# Convert and submit  
report = uut.to_report()  
api.report.send_uut_report(report)  
print(f"Report submitted for {uut.serial_number}")
```

Manual Report Creation (Advanced)

If you need fine-grained control, create reports directly:

```
from pywats.models import UUTReport, UUTInfo, SequenceCall, NumericStep  
from datetime import datetime, timezone  
  
# Create report structure manually  
report = UUTReport()  
report.info = UUTInfo(  
    part_number="PART-001",  
    serial_number="SN-001",  
    revision="A",  
    operator="Operator Name",  
    start_date_time=datetime.now(timezone.utc)  
)  
  
# Create root sequence  
root = SequenceCall(name="MainSequence")  
report.add_step(root)  
  
# Add step manually  
step = NumericStep(  
    step_type="ET_NLT",  
    name="Voltage",  
    value=5.0,  
    unit="V",  
    comp_op="GELE",  
    low_limit=4.5,  
    high_limit=5.5,  
    status="Passed"  
)  
root.add_step(step)  
  
# Submit  
api.report.send_uut_report(report)
```

 **Warning:** Manual creation requires understanding the internal structure. Use `TestUUT` factory instead.

UUR Reports (Repairs)

Creating Repair Reports

```
from pywats.models import UURReport

# Option 1: From existing UUT report
uur = api.report.create_uur_report(
    from_uut_report=uut_report,
    failure_category=500, # Repair category code
    failure_code=501,      # Specific failure code
    description="Replaced capacitor C15",
    action_taken="Component replacement",
    operator="Repair Tech"
)
api.report.send_uur_report(uur)

# Option 2: From part number and process
uur = api.report.create_uur_from_part_and_process(
    part_number="PART-001",
    serial_number="SN-001",
    revision="A",
    process_code=100,
    failure_category=500,
    failure_code=501,
    description="Failed power test",
    operator="Repair Tech"
)
api.report.send_uur_report(uur)
```

Querying Reports

The report query API uses OData filters for flexible querying. The helper methods provide a simpler interface for common queries.

Using OData Filters

```
from pywats.domains.report import ReportType

# Filter by part number
headers = api.report.query_uut_headers(
    odata_filter="partNumber eq 'PART-001'"
)

# Filter by serial number
headers = api.report.query_uut_headers(
    odata_filter="serialNumber eq 'SN-12345'"
)

# Filter by result
headers = api.report.query_uut_headers(
    odata_filter="result eq 'Passed'",
    top=100
)

# Combined filters
headers = api.report.query_uut_headers(
    odata_filter="partNumber eq 'PART-001' and result eq 'Failed'",
    top=100,
    orderby="start desc"
)

# Query repair (UUR) reports
repairs = api.report.query_uur_headers(
    odata_filter="serialNumber eq 'SN-12345'"
)

# Unified query using ReportType enum
headers = api.report.query_headers(
    report_type=ReportType.UUT,
    odata_filter="serialNumber eq 'SN-12345'"
)
```

Convenience Methods

```
# Get reports by serial number
headers = api.report.get_headers_by_serial("SN-12345")

# Get reports by part number
headers = api.report.get_headers_by_part_number("PART-001")

# Get reports by date range
from datetime import datetime, timedelta
start = datetime.now() - timedelta(days=7)
end = datetime.now()
headers = api.report.get_headers_by_date_range(start, end)

# Get recent reports
headers = api.report.get_recent_headers(count=50)

# Get today's reports
headers = api.report.get_todays_headers()

# Load full report by UUID
report = api.report.get_uut_report("report-uuid-here")
```

Expanded Fields

```
# Include sub-units in response
headers = api.report.query_uut_headers(
    odata_filter="serialNumber eq 'SN-12345'",
    expand=["subUnits"]
)

# Include miscellaneous info
headers = api.report.query_uut_headers(
    odata_filter="partNumber eq 'PART-001'",
    expand=["miscInfo"]
)
```

Best Practices

1. Always Use TestUUT Factory

```
# ✓ RECOMMENDED
uut = TestUUT(...)
root = uut.get_root()
root.add_numeric_step(...)

# X AVOID - Manual creation is error-prone
report = UUTReport()
report.info = UUTInfo(...)
# ... lots of boilerplate ...
```

2. Use Appropriate Step Types

```
# ✓ Good - use specific types
root.add_numeric_step("Voltage", value=5.0, ...)
root.add_boolean_step("LED Test", value=True, ...)

# X Avoid - generic steps when specific ones exist
root.add_string_step("Voltage", value="5.0", ...) # Should be numeric
```

3. Organize with Sequences

```
# ✓ Good - grouped logically
power_tests = root.add_sequence("Power Tests")
power_tests.add_numeric_step(...)
power_tests.add_numeric_step(...)

io_tests = root.add_sequence("I/O Tests")
io_tests.add_boolean_step(...)

# X Avoid - flat structure
root.add_numeric_step("Power 1", ...)
root.add_numeric_step("Power 2", ...)
root.add_boolean_step("I/O 1", ...) # Mixed without grouping
```

4. Set Meaningful Status

```
# ✓ Good - clear status
root.add_numeric_step(..., status="Passed")
root.add_boolean_step(..., status="Failed")
root.add_string_step(..., status="Done") # For informational steps

# ✗ Avoid - wrong status
root.add_numeric_step(..., status="Done") # Should be Passed/Failed
```

5. Include Units

```
# ✓ Good - includes units
root.add_numeric_step("Voltage", value=5.0, unit="V", ...)

# ✗ Avoid - missing units
root.add_numeric_step("Voltage", value=5.0, ...) # What unit?
```

Common Patterns

Pattern 1: Test with Retry Logic

```
def run_test_with_retry(uut: TestUUT, max_retries=3):
    root = uut.get_root()

    for attempt in range(max_retries):
        result = perform_test() # Your test logic

        if result.passed:
            root.add_boolean_step(
                f"Test (attempt {attempt + 1})",
                value=True,
                status="Passed"
            )
            break
        else:
            root.add_boolean_step(
                f"Test (attempt {attempt + 1})",
                value=False,
                status="Failed"
            )

    return uut.to_report()
```

Pattern 2: Conditional Testing

```
def conditional_test(uut: TestUUT):
    root = uut.get_root()

    # Initial test
    voltage = measure_voltage()
    root.add_numeric_step("Initial Voltage", voltage, "V", ...)

    # Only do burn-in if voltage is good
    if 4.5 <= voltage <= 5.5:
        burnin_seq = root.add_sequence("Burn-In Test")
        # ... burn-in steps ...
    else:
        root.add_string_step("Burn-In", "Skipped - voltage out of range", "Skipped")

    return uut.to_report()
```

Pattern 3: Data-Driven Testing

```
def data_driven_test(uut: TestUUT, test_points):
    root = uut.get_root()

    measurements = []
    for point in test_points:
        value = measure_at_point(point)
        measurements.append({
            "name": point.name,
            "value": value,
            "unit": point.unit,
            "comp_op": "GELE",
            "low_limit": point.min,
            "high_limit": point.max,
            "status": "Passed" if point.min <= value <= point.max else "Failed"
        })

    root.add_multi_numeric_step(
        name="Test Points",
        measurements=measurements,
        status="Passed" if all(m["status"] == "Passed" for m in measurements) else "Failed"
    )

    return uut.to_report()
```

Troubleshooting

Factory Methods Not Working?

```
# Make sure you import from tools
from pywats.tools.test_uut import TestUUT # ✓ Correct

from pywats.models import UUTReport # X Wrong - this is the model, not factory
```

Steps Not Appearing in Report?

```
# Make sure you call methods on the sequence, not the factory
root = uut.get_root() # Get the root sequence
root.add_numeric_step(...) # ✓ Correct

uut.add_numeric_step(...) # X Wrong - call on root
```

Report Submission Failing?

```
# Convert to report before sending
report = uut.to_report() # Don't forget this!
api.report.send_uut_report(report)

# Not:
api.report.send_uut_report(uut) # X Wrong - uut is factory, not report
```

Related Documentation

- Production Module - For serial number management before testing
- Product Module - For product/revision setup
- Architecture - Overall system design

ReportBuilder - Simple Report Building for pyWATS

Overview

The **ReportBuilder** is a forgiving, LLM-friendly tool for creating WATS test reports with minimal complexity. It's designed for:

- **Converter scripts** that parse test data from various formats
- **LLM-generated code** that needs to build reports without deep understanding
- **Quick prototyping** where you don't want to think about report structure
- **Situations with messy data** that needs graceful handling

Philosophy

- **If it can be inferred, it will be** - Step types, comparison operators, and statuses are automatically determined
- **If it's missing, use sensible defaults** - You only specify what you have
- **Never fail on missing metadata** - Do the best with what's provided
- **Support both flat and hierarchical structures** - Automatically organize by groups

Quick Start

Simple Example

```
from pywats.tools import ReportBuilder

# Create builder
builder = ReportBuilder(
    part_number="MODULE-100",
    serial_number="MOD-2025-001"
)

# Add steps - it figures out the types
builder.add_step("Voltage Test", 5.02, unit="V", low_limit=4.5, high_limit=5.5)
builder.add_step("Power OK", True)
builder.add_step("Serial Read", "ABC123")

# Build and submit
report = builder.build()
api.report.submit_report(report)
```

One-Liner with quick_report()

```
from pywats.tools import quick_report

steps = [
    {"name": "Voltage", "value": 5.0, "unit": "V", "low_limit": 4.5, "high_limit": 5.5},
    {"name": "Current", "value": 1.2, "unit": "A"},
    {"name": "Status", "value": True}
]

report = quick_report("PN-001", "SN-001", steps)
api.report.submit_report(report)
```

Key Features

1. Automatic Type Inference

The builder automatically determines the correct step type based on your data:

```
builder.add_step("Boolean Test", True)          # → Boolean step
builder.add_step("Numeric Test", 5.0)            # → Numeric step
builder.add_step("String Test", "ABC")           # → String step
builder.add_step("Multi-Numeric", [1, 2, 3])     # → Multi-numeric step
```

2. Smart Status Calculation

Status is automatically calculated from limits unless you override it:

```
# Auto-calculated as "P" (in range)
builder.add_step("Test1", 5.0, low_limit=4.0, high_limit=6.0)

# Auto-calculated as "F" (out of range)
builder.add_step("Test2", 10.0, low_limit=4.0, high_limit=6.0)

# Manual override
builder.add_step("Test3", 5.0, status="F") # Force fail
```

3. Flexible Data Handling

Works with messy, real-world data:

```
# Limits as strings (auto-converted)
builder.add_step("Test", "5.02", low_limit="4.5", high_limit="5.5")

# Various status formats
builder.add_step("Test1", 5.0, status="PASS")
builder.add_step("Test2", 5.0, status="P")
builder.add_step("Test3", 5.0, status="Passed")
builder.add_step("Test4", 5.0, status=True)

# Boolean from string
builder.add_step("Test", "TRUE") # Converts to boolean
```

4. Automatic Grouping/Sequences

Create hierarchical reports by specifying groups:

```
builder = ReportBuilder("PN-001", "SN-001")

# Power tests group
builder.add_step("VCC", 3.3, unit="V", group="Power Tests")
builder.add_step("VDD", 1.8, unit="V", group="Power Tests")

# Communication tests group
builder.add_step("UART", True, group="Communication")
builder.add_step("I2C", True, group="Communication")

# Build creates sequences automatically
report = builder.build()
```

Result:

```
MainSequence
├── Power Tests (sequence)
│   ├── VCC
│   └── VDD
└── Communication (sequence)
    ├── UART
    └── I2C
```

API Reference

ReportBuilder

```
class ReportBuilder:
    def __init__(
        self,
        part_number: str,
        serial_number: str,
        revision: str = "A",
        operator: Optional[str] = None,
        station: Optional[str] = None,
        location: Optional[str] = None,
        process_code: int = 10,
        result: Optional[str] = None,
        start_time: Optional[datetime] = None,
        purpose: Optional[str] = None
    )
```

Parameters:

- `part_number` (required): Part number
- `serial_number` (required): Serial number
- `revision` : Revision (default: "A")
- `operator` : Operator name (default: "Converter")
- `station` : Station name
- `location` : Location
- `process_code` : Operation type code (default: 10 = SW Debug)
- `result` : Overall result "P" or "F" (auto-calculated if None)
- `start_time` : Start time (defaults to now)
- `purpose` : Test purpose

add_step()

```
def add_step(
    self,
    name: str,
    value: Any = None,
    unit: Optional[str] = None,
    low_limit: Optional[Union[float, str]] = None,
    high_limit: Optional[Union[float, str]] = None,
    status: Optional[str] = None,
    group: Optional[str] = None,
    comp_op: Optional[Union[CompOp, str]] = None,
    **kwargs
) -> "ReportBuilder"
```

Type Inference:

- bool / "TRUE" / "FALSE" / "PASS" / "FAIL" → Boolean step
- float / int with limits → Numeric limit test
- float / int without limits → Numeric log
- str → String step
- list[float] / list[int] → Multi-numeric step
- list[bool] → Multi-boolean step
- list[str] → Multi-string step

Parameters:

- name (required): Step name
- value : Measured value (can be anything)
- unit : Unit of measurement (e.g., "V", "A", "°C")
- low_limit : Lower limit (for numeric tests)
- high_limit : Upper limit (for numeric tests)
- status : Status override ("P", "F", "Passed", "Failed", etc.)
- group : Group/sequence name (creates hierarchy)
- comp_op : Comparison operator (auto-inferred if not specified)
- **kwargs : Additional metadata stored for debugging

Returns: Self (for method chaining)

[add_step_from_dict\(\)](#)

```
def add_step_from_dict(
    self,
    data: Dict[str, Any],
    name_key: str = "name",
    value_key: str = "value",
    unit_key: str = "unit",
    low_limit_key: str = "low_limit",
    high_limit_key: str = "high_limit",
    status_key: str = "status",
    group_key: str = "group"
) -> "ReportBuilder"
```

Add a step from a dictionary with flexible key mapping. Automatically tries common variations:

- name: name , Name , TestName , test_name , Step
- value: value , Value , MeasuredValue , Result
- unit: unit , Unit , Units , UOM
- low_limit: low_limit , LowLimit , Low , MinLimit , min
- high_limit: high_limit , HighLimit , High , MaxLimit , max
- status: status , Status , Result , Pass
- group: group , Group , Sequence , TestGroup

[add_misc_info\(\)](#)

```
def add_misc_info(
    self,
    description: str,
    text: str
) -> "ReportBuilder"
```

Add searchable metadata to the report header.

[add_sub_unit\(\)](#)

```
def add_sub_unit(
    self,
    part_type: str,
    part_number: Optional[str] = None,
    serial_number: Optional[str] = None,
    revision: Optional[str] = None
) -> "ReportBuilder"
```

Add a sub-unit (component) to the report.

build()

```
def build(self) -> UUTReport
```

Build the final UUTReport. This method:

1. Creates UUTReport with header info
2. Groups steps by sequence (if groups specified)
3. Adds all steps in correct order
4. Sets overall result based on step statuses
5. Adds misc info and sub-units

Returns: UUTReport ready to submit

quick_report()

```
def quick_report(  
    part_number: str,  
    serial_number: str,  
    steps: List[Dict[str, Any]],  
    **kwargs  
) -> UUTReport
```

Create a report from a list of step dictionaries in one call. Perfect for LLM-generated code.

Usage Examples

Example 1: Simple Converter

```
from pywats.tools import ReportBuilder

def convert_my_format(file_path):
    # Parse your file format
    data = parse_file(file_path)

    # Create builder
    builder = ReportBuilder(
        part_number=data["part_number"],
        serial_number=data["serial_number"]
    )

    # Add all tests
    for test in data["tests"]:
        builder.add_step(
            name=test["name"],
            value=test["value"],
            unit=test.get("unit"),
            low_limit=test.get("low_limit"),
            high_limit=test.get("high_limit")
        )

    # Done!
    return builder.build()
```

Example 2: From CSV Data

```
import csv
from pywats.tools import ReportBuilder

def convert_csv(csv_file):
    with open(csv_file) as f:
        reader = csv.DictReader(f)

        builder = ReportBuilder("PN-FROM-CSV", "SN-FROM-CSV")

        for row in reader:
            builder.add_step_from_dict(
                row,
                name_key="TestName",
                value_key="MeasuredValue",
                unit_key="Unit",
                low_limit_key="LowLimit",
                high_limit_key="HighLimit"
            )

    return builder.build()
```

Example 3: With Metadata

```
builder = ReportBuilder(
    part_number="ASSEMBLY-500",
    serial_number="ASM-2025-0123",
    operator="Production Line 3",
    station="Final Test"
)

# Add tests
builder.add_step("Visual Inspection", "PASS")
builder.add_step("Voltage", 12.05, unit="V", low_limit=11.5, high_limit=12.5)

# Add metadata
builder.add_misc_info("Batch Number", "BATCH-2025-Q1-042")
builder.add_misc_info("Temperature", "22°C")

# Add components
builder.add_sub_unit(
    part_type="Power Supply",
    part_number="PSU-24V-100W",
    serial_number="PSU-001"
)

report = builder.build()
```

Example 4: ICT Converter

```
def convert_ict_data(ict_file):
    data = parse_ict_file(ict_file)

    builder = ReportBuilder(
        part_number=data["pn"],
        serial_number=data["sn"],
        operator=data["operator"]
    )

    # Group by test type
    for resistance in data["resistances"]:
        builder.add_step(
            name=resistance["designator"],
            value=resistance["measured"],
            unit="Ω",
            low_limit=resistance["min"],
            high_limit=resistance["max"],
            group="Resistance Tests"
        )

    for capacitor in data["capacitors"]:
        builder.add_step(
            name=capacitor["designator"],
            value=capacitor["measured"],
            unit="µF",
            low_limit=capacitor["min"],
            high_limit=capacitor["max"],
            group="Capacitance Tests"
        )

    return builder.build()
```

LLM Integration Guide

For LLM Monitoring/Autocorrection

When monitoring converter code, check for these patterns:

Good Pattern:

```
builder = ReportBuilder(pn, sn)
builder.add_step("Test", value, unit="V", low_limit=4.5, high_limit=5.5)
report = builder.build()
```

Avoid Direct UUTReport Construction:

```
# Too complex for LLMs to get right
report = UUTReport(pn=pn, sn=sn, ...)
root = report.get_root_sequence_call()
root.add_numeric_step(...) # Easy to mess up parameters
```

For LLM Implementation of New Converters

Template for LLMs:

```
from pywats.tools import ReportBuilder

def convert_my_format(file_path):
    # 1. Parse file (this is the only custom part)
    data = parse_your_format(file_path)

    # 2. Create builder
    builder = ReportBuilder(
        part_number=data["part_number"],
        serial_number=data["serial_number"]
    )

    # 3. Add steps (one line per test)
    for test in data["tests"]:
        builder.add_step(
            name=test["name"],
            value=test["value"],
            unit=test.get("unit"),
            low_limit=test.get("low_limit"),
            high_limit=test.get("high_limit"),
            group=test.get("group") # Optional
        )

    # 4. Build and return
    return builder.build()
```

LLM should only customize: The `parse_your_format()` function to extract data from the specific file format.

Everything else is standard.

Advanced Features

Multi-Value Steps

```
# Multi-numeric (array of numbers)
builder.add_step(
    "Calibration Points",
    [1.2, 1.3, 1.1, 1.25, 1.15],
    unit="mV",
    low_limit=1.0,
    high_limit=1.5
)

# Multi-boolean (array of pass/fail)
builder.add_step("Pin Tests", [True, True, False, True])

# Multi-string (array of strings)
builder.add_step("Serial Numbers", ["ABC", "DEF", "GHI"])
```

Custom Comparison Operators

```
from pywats.domains.report.report_models.uut.steps.comp_operator import CompOp

builder.add_step(
    "Exact Value",
    5.0,
    low_limit=5.0,
    comp_op=CompOp.EQ # Must equal exactly
)

builder.add_step(
    "Greater Than",
    10.0,
    low_limit=5.0,
    comp_op=CompOp.GT # Must be > 5.0 (not >=)
)
```

Failed Reports

```
builder = ReportBuilder("PN", "SN")

builder.add_step("Test1", 5.0, low_limit=4.0, high_limit=6.0) # Pass
builder.add_step("Test2", 10.0, low_limit=4.0, high_limit=6.0) # FAIL

report = builder.build()
# report.result will be "F" because Test2 failed
```

Comparison with Direct API

Before (Complex):

```
from pywats.domains.report.report_models import UUTReport
from pywats.domains.report.report_models.uut.steps.comp_operator import CompOp

report = UUTReport(
    pn="PN-001",
    sn="SN-001",
    rev="A",
    process_code=10,
    station_name="Station1",
    result="P",
    start=datetime.now()
)

root = report.get_root_sequence_call()

# Need to know exact parameters, types, and comparison operators
root.add_numeric_step(
    name="Voltage",
    value=5.02,
    unit="V",
    comp_op=CompOp.GELE,
    low_limit=4.5,
    high_limit=5.5,
    status="P"
)
```

After (Simple):

```
from pywats.tools import ReportBuilder

builder = ReportBuilder("PN-001", "SN-001")

# Just provide data - everything else is inferred
builder.add_step("Voltage", 5.02, unit="V", low_limit=4.5, high_limit=5.5)

report = builder.build()
```

Best Practices

1. **Use ReportBuilder for converters** - It handles edge cases better than manual construction
2. **Let it infer types** - Don't overthink numeric vs string vs boolean
3. **Use groups for organization** - Makes reports easier to read in WATS

4. **Add misc_info for metadata** - Searchable in WATS
5. **Let status auto-calculate** - Only override if you have explicit pass/fail from source

Troubleshooting

Problem: Status always "P" even with failures

Solution: Make sure limits are numeric (not strings), or provide explicit status:

```
# Auto-fails if out of range
builder.add_step("Test", 10.0, low_limit=4.0, high_limit=6.0)

# Or explicit
builder.add_step("Test", 10.0, status="F")
```

Problem: Wrong step type created

Solution: Ensure value is correct Python type:

```
# Boolean
builder.add_step("Test", True) # ✓
builder.add_step("Test", "true") # X (becomes string step)

# Numeric
builder.add_step("Test", 5.0) # ✓
builder.add_step("Test", "5.0") # ? (auto-converted, but better to convert yourself)
```

Problem: Can't find data in dictionary

Solution: Use `add_step_from_dict()` with explicit key names:

```
builder.add_step_from_dict(
    data,
    name_key="TestName", # Specify exact keys
    value_key="MeasuredValue"
)
```

See Also

- Report Module Documentation
- Converter Template
- Examples

Asset Module Usage Guide

Overview

The Asset module manages test fixtures, and other manufacturing assets in WATS. Assets can be hierarchical (e.g., stations containing instruments) and track calibration, maintenance, and usage history.

Quick Start

```
from pywats import pyWATS

api = pyWATS(base_url="https://wats.example.com", token="credentials")

# Get all assets
assets = api.asset.get_assets()

# Get specific asset
asset = api.asset.get_asset(asset_id)

# Get assets by type
stations = api.asset.get_assets_by_type("Station")
```

Asset Types

Assets are categorized by type. Common types include:

Type	Description	Examples
Station	Test station or workstation	ICT-01, FVT-STATION-01
Instrument	Measurement equipment	DMM, Oscilloscope, Power Supply
Fixture	Test fixtures	Bed-of-nails, Custom fixture
Software	Software tools	LabVIEW RT, TestStand
Tool	Hand tools or equipment	Soldering iron, Torque wrench

Basic Operations

1. Get Assets

```
# Get all assets (summary view)
assets = api.asset.get_assets()

for asset in assets:
    print(f"{asset.name}: {asset.asset_type}")
    print(f" ID: {asset.asset_id}")
    print(f" Serial: {asset.serial_number}")
    print(f" State: {asset.state}")

# Get full asset details (with nested data)
asset = api.asset.get_asset_full(asset_id)
print(f"Children: {len(asset.children)} if asset.children else 0")
```

2. Get Asset by ID

```
# Get specific asset
asset = api.asset.get_asset(asset_id)

if asset:
    print(f"Name: {asset.name}")
    print(f"Type: {asset.asset_type}")
    print(f"Serial Number: {asset.serial_number}")
    print(f"State: {asset.state}")
    print(f"Location: {asset.location}")
```

3. Get Assets by Type

```
# Get all assets of a specific type
stations = api.asset.get_assets_by_type("Station")
instruments = api.asset.get_assets_by_type("Instrument")
fixtures = api.asset.get_assets_by_type("Fixture")

print(f"Found {len(stations)} stations")
for station in stations:
    print(f" - {station.name} ({station.serial_number})")
```

4. Create Asset

```
from pywats.domains.asset import AssetState

# Create new asset
asset = api.asset.create_asset(
    name="DMM-001",
    asset_type="Instrument",
    serial_number="DMM2024-12345",
    state=AssetState.ACTIVE,
    description="Keithley 2000 Digital Multimeter",
    location="Lab A"
)

print(f"Created asset: {asset.name} (ID: {asset.asset_id})")
```

5. Update Asset

```
# Get asset, modify, update
asset = api.asset.get_asset(asset_id)
asset.description = "Updated description"
asset.location = "Lab B"
asset.state = AssetState.IN_CALIBRATION

updated = api.asset.update_asset(asset)
print(f"Updated: {updated.name}")
```

6. Delete Asset

```
# Delete an asset (use with caution)
success = api.asset.delete_asset(asset_id)
if success:
    print("Asset deleted")
```

Asset Hierarchy

Assets can have parent-child relationships (e.g., instruments inside a station).

1. Get Child Assets

```
# Get children of an asset
children = api.asset.get_child_assets(parent_asset_id)

print(f"Station has {len(children)} instruments:")
for child in children:
    print(f" - {child.name} ({child.asset_type})")
```

2. Create Child Asset

```
# Create asset as child of another
instrument = api.asset.create_asset(
    name="DMM-Internal",
    asset_type="Instrument",
    serial_number="DMM-INT-001",
    state=AssetState.ACTIVE,
    parent_id=station_asset_id # Parent asset
)
```

3. Move Asset (Change Parent)

```
# Move asset to new parent
asset = api.asset.get_asset(asset_id)
asset.parent_id = new_parent_id
api.asset.update_asset(asset)
```

Asset States

```
from pywats.domains.asset import AssetState

# Available states
AssetState.ACTIVE          # In use, operational
AssetState.INACTIVE         # Not in use
AssetState.IN_CALIBRATION   # Being calibrated
AssetState.IN_REPAIR         # Being repaired
AssetState.RETIRED           # End of life
```

State Transitions

```
# Example: Send asset for calibration
asset = api.asset.get_asset(asset_id)
asset.state = AssetState.IN_CALIBRATION
api.asset.update_asset(asset)

# After calibration complete
asset.state = AssetState.ACTIVE
asset.calibration_date = datetime.now()
asset.next_calibration_date = datetime.now() + timedelta(days=365)
api.asset.update_asset(asset)
```

Asset Tags

Assets can have tags (key-value metadata):

1. Get Asset Tags

```
# Get tags for an asset
tags = api.asset.get_asset_tags(asset_id)
for tag in tags:
    print(f"{tag.key}: {tag.value}")
```

2. Set Asset Tags

```
from pywats.shared import Setting

# Set tags (replaces all existing)
api.asset.set_asset_tags(asset_id, [
    Setting(key="Manufacturer", value="Keithley"),
    Setting(key="Model", value="2000"),
    Setting(key="CalibrationInterval", value="365")
])
```

3. Add Single Tag

```
# Add/update single tag
api.asset.add_asset_tag(asset_id, "Location", "Lab A")
```

Calibration Tracking

1. Track Calibration Dates

```
from datetime import datetime, timedelta

# Update calibration info
asset = api.asset.get_asset(asset_id)
asset.calibration_date = datetime.now()
asset.next_calibration_date = datetime.now() + timedelta(days=365)
api.asset.update_asset(asset)
```

2. Find Assets Due for Calibration

```
from datetime import datetime

# Get all instruments
instruments = api.asset.get_assets_by_type("Instrument")

# Find those due for calibration
due_soon = []
for instr in instruments:
    if instr.next_calibration_date:
        days_until = (instr.next_calibration_date - datetime.now()).days
        if days_until <= 30:
            due_soon.append({
                "asset": instr,
                "days_until": days_until
            })

print("Assets due for calibration within 30 days:")
for item in due_soon:
    print(f" {item['asset'].name}: {item['days_until']} days")
```

Common Patterns

Pattern 1: Station Setup

```
from pywats.domains.asset import AssetState

# 1. Create station
station = api.asset.create_asset(
    name="ICT-STATION-01",
    asset_type="Station",
    serial_number="ICT-2024-001",
    state=AssetState.ACTIVE,
    description="In-Circuit Test Station 1",
    location="Production Floor"
)

# 2. Add instruments to station
dmm = api.asset.create_asset(
    name="ICT-01-DMM",
    asset_type="Instrument",
    serial_number="DMM-001",
    state=AssetState.ACTIVE,
    description="Station DMM",
    parent_id=station.asset_id
)

power_supply = api.asset.create_asset(
    name="ICT-01-PSU",
    asset_type="Instrument",
    serial_number="PSU-001",
    state=AssetState.ACTIVE,
    description="Station Power Supply",
    parent_id=station.asset_id
)

# 3. Tag the station
api.asset.set_asset_tags(station.asset_id, [
    Setting(key="Line", value="Production Line 1"),
    Setting(key="Process", value="ICT"),
    Setting(key="Shift", value="All")
])
```

Pattern 2: Asset Inventory Report

```
def generate_inventory_report():
    """Generate asset inventory by type"""
    assets = api.asset.get_assets()

    # Group by type
    by_type = {}
    for asset in assets:
        asset_type = asset.asset_type or "Unknown"
        if asset_type not in by_type:
            by_type[asset_type] = []
        by_type[asset_type].append(asset)

    # Print report
    print("=" * 50)
    print("ASSET INVENTORY REPORT")
    print("=" * 50)

    for asset_type, items in sorted(by_type.items()):
        print(f"\n{asset_type}: {len(items)} items")
        for item in items:
            state = item.state.name if item.state else "Unknown"
            print(f"  - {item.name} ({item.serial_number}) [{state}]")

    print("\nTotal: {len(assets)} assets")
```

Pattern 3: Calibration Management

```
from datetime import datetime, timedelta
from pywats.domains.asset import AssetState

def send_for_calibration(asset_id: str):
    """Mark asset as in calibration"""
    asset = api.asset.get_asset(asset_id)
    asset.state = AssetState.IN_CALIBRATION
    api.asset.update_asset(asset)
    print(f"{asset.name} sent for calibration")

def complete_calibration(asset_id: str, interval_days: int = 365):
    """Complete calibration and update dates"""
    asset = api.asset.get_asset(asset_id)
    asset.state = AssetState.ACTIVE
    asset.calibration_date = datetime.now()
    asset.next_calibration_date = datetime.now() + timedelta(days=interval_days)
    api.asset.update_asset(asset)
    print(f"{asset.name} calibration complete, next due: {asset.next_calibration_date}")

def get_calibration_schedule():
    """Get upcoming calibration schedule"""
    instruments = api.asset.get_assets_by_type("Instrument")

    schedule = []
    for instr in instruments:
        if instr.next_calibration_date and instr.state == AssetState.ACTIVE:
            schedule.append({
                "name": instr.name,
                "serial": instr.serial_number,
                "due_date": instr.next_calibration_date,
                "days_remaining": (instr.next_calibration_date - datetime.now()).days
            })

    # Sort by due date
    schedule.sort(key=lambda x: x["due_date"])
    return schedule
```

Pattern 4: Asset Search

```
def find_assets(
    name_contains: str = None,
    asset_type: str = None,
    state: AssetState = None,
    location: str = None
):
    """Search assets with filters"""
    assets = api.asset.get_assets()

    results = []
    for asset in assets:
        # Apply filters
        if name_contains and name_contains.lower() not in (asset.name or "").lower():
            continue
        if asset_type and asset.asset_type != asset_type:
            continue
        if state and asset.state != state:
            continue
        if location and location.lower() not in (asset.location or "").lower():
            continue

        results.append(asset)

    return results

# Usage
dmms = find_assets(name_contains="DMM", asset_type="Instrument")
active_stations = find_assets(asset_type="Station", state=AssetState.ACTIVE)
lab_equipment = find_assets(location="Lab")
```

Asset Model Reference

Asset Fields

Field	Type	Description
asset_id	UUID	Unique identifier
name	str	Asset name
asset_type	str	Type (Station, Instrument, etc.)
serial_number	str	Serial number
description	str	Description
state	AssetState	Current state
location	str	Physical location
parent_id	UUID	Parent asset ID (for hierarchy)
calibration_date	datetime	Last calibration date
next_calibration_date	datetime	Next calibration due
children	List[Asset]	Child assets (when loaded)
tags	List[Setting]	Key-value tags

AssetState Enum

Value	Description
ACTIVE	In use, operational
INACTIVE	Not currently in use
IN_CALIBRATION	Being calibrated
IN_REPAIR	Under repair
RETired	End of life

Best Practices

1. Use Meaningful Names

```
# Good - descriptive names
"ICT-STATION-01"
"FVT-DMM-KEITHLEY"
"FIXTURE-PCBA-V2"

# Avoid - unclear names
"ASSET1"
"INST"
```

2. Track Serial Numbers

```
# Always include serial numbers for traceability
asset = api.asset.create_asset(
    name="DMM-001",
    serial_number="KTH-2000-12345", # Include manufacturer serial
    ...
)
```

3. Use Tags for Custom Data

```
# Tags for flexible metadata
api.asset.set_asset_tags(asset_id, [
    Setting(key="Manufacturer", value="Keithley"),
    Setting(key="Model", value="2000"),
    Setting(key="PurchaseDate", value="2024-01-15"),
    Setting(key="WarrantyExpires", value="2027-01-15"),
    Setting(key="CostCenter", value="PROD-001")
])
```

4. Maintain Hierarchy

```
# Organize assets hierarchically
# Station -> Instruments -> Components

station = api.asset.create_asset(name="FVT-01", asset_type="Station", ...)
dmm = api.asset.create_asset(name="FVT-01-DMM", parent_id=station.asset_id, ...)
scope = api.asset.create_asset(name="FVT-01-SCOPE", parent_id=station.asset_id, ...)
```

5. Track Calibration

```
# Set calibration dates when creating instruments
from datetime import datetime, timedelta

instrument = api.asset.create_asset(
    name="DMM-001",
    asset_type="Instrument",
    calibration_date=datetime.now(),
    next_calibration_date=datetime.now() + timedelta(days=365),
    ...
)
```

Troubleshooting

Asset Not Found

```
asset = api.asset.get_asset(asset_id)
if not asset:
    print(f"Asset {asset_id} not found")
    # Try searching by name
    assets = api.asset.get_assets()
    matches = [a for a in assets if "DMM" in a.name]
```

Parent-Child Issues

```
# Verify parent exists before creating child
parent = api.asset.get_asset(parent_id)
if not parent:
    print("Parent asset not found, create it first")
```

State Transitions

```
# Some state transitions may be restricted
# Always check current state before changing
asset = api.asset.get_asset(asset_id)
if asset.state == AssetState.RETIRED:
    print("Cannot modify retired asset")
```

Related Documentation

- Product Module - Managing products tested by assets

- Production Module - Manufacturing units at stations
 - Report Module - Test reports from stations
-

Source: [docs/usage/process-module.md](#)

Process Module Usage Guide

Overview

The Process module provides access to process/operation definitions in WATS. Processes define the types of operations that can be performed during manufacturing:

- **Test Operations** - End-of-line tests, ICT tests, functional tests
- **Repair Operations** - Repair, RMA repair, rework
- **WIP Operations** - Work-in-progress tracking, assembly steps

Quick Start

```
from pywats import pyWATS

api = pyWATS(base_url="https://wats.example.com", token="credentials")

# Get all processes
processes = api.process.get_processes()

# Get a specific test operation
test_op = api.process.get_test_operation(100) # By code
test_op = api.process.get_test_operation("End of line test") # By name

# Validate a process code
if api.process.is_valid_test_operation(100):
    print("Valid test operation")
```

Process Types

Type	Flag	Typical Codes	Examples
Test	is_test_operation	100-499	End of line test, ICT, FCT
Repair	is_repair_operation	500-599	Repair, RMA Repair
WIP	is_wip_operation	200-299	Assembly, Inspection

```
# Check process type
if process.is_test_operation:
    print("This is a test operation")
elif process.is_repair_operation:
    print("This is a repair operation")
elif process.is_wip_operation:
    print("This is a WIP operation")
```

Caching

The Process service maintains an in-memory cache to minimize API calls. By default, the cache refreshes every 5 minutes.

```
# Configure cache refresh interval (seconds)
api.process.refresh_interval = 600 # 10 minutes

# Force cache refresh
api.process.refresh()

# Check cache status
print(f"Last refresh: {api.process.last_refresh}")
print(f"Refresh interval: {api.process.refresh_interval}s")
```

Basic Operations

1. Get All Processes

```
# Get all processes (cached)
processes = api.process.get_processes()

for proc in processes:
    print(f"{proc.code}: {proc.name}")
    if proc.is_test_operation:
        print("  Type: Test Operation")
    elif proc.is_repair_operation:
        print("  Type: Repair Operation")
    elif proc.is_wip_operation:
        print("  Type: WIP Operation")
```

2. Get Processes by Type

```
# Get test operations only
test_ops = api.process.get_test_operations()
for op in test_ops:
    print(f"{op.code}: {op.name}")

# Get repair operations only
repair_ops = api.process.get_repair_operations()

# Get WIP operations only
wip_ops = api.process.get_wip_operations()
```

3. Get Specific Process

```
# By code (int)
process = api.process.get_process(100)

# By name (str, case-insensitive)
process = api.process.get_process("End of line test")

if process:
    print(f"Found: {process.code} - {process.name}")
else:
    print("Process not found")
```

4. Get Specific Operation Types

```
# Get test operation (returns None if not a test operation)
test_op = api.process.get_test_operation(100)
test_op = api.process.get_test_operation("ICT Test")

# Get repair operation (returns None if not a repair operation)
repair_op = api.process.get_repair_operation(500)
repair_op = api.process.get_repair_operation("Repair")

# Get WIP operation
wip_op = api.process.get_wip_operation(200)
wip_op = api.process.get_wip_operation("Assembly")
```

Validation Helpers

Validate Process Codes

```
# Validate test operation code
if api.process.is_valid_test_operation(100):
    print("Code 100 is a valid test operation")
else:
    print("Code 100 is NOT a valid test operation")

# Validate repair operation code
if api.process.is_valid_repair_operation(500):
    print("Code 500 is a valid repair operation")

# Validate WIP operation code
if api.process.is_valid_wip_operation(200):
    print("Code 200 is a valid WIP operation")
```

Get Default Codes

```
# Get default test code (first available or 100 as fallback)
default_test = api.process.get_default_test_code()
print(f"Default test code: {default_test}")

# Get default repair code (first available or 500 as fallback)
default_repair = api.process.get_default_repair_code()
print(f"Default repair code: {default_repair}")
```

Common Patterns

Pattern 1: Process Validation Before Report Submission

```
def validate_report_process(api, process_code):
    """Validate process code before submitting a report"""

    if not api.process.is_valid_test_operation(process_code):
        raise ValueError(f"Invalid test operation code: {process_code}")

    process = api.process.get_test_operation(process_code)
    return process.name

# Usage
try:
    process_name = validate_report_process(api, 100)
    print(f"Submitting report for: {process_name}")
except ValueError as e:
    print(f"Error: {e}")
```

Pattern 2: Display Available Operations

```
def display_available_operations(api):
    """Display all available operations by type"""

    print("== TEST OPERATIONS ==")
    for op in api.process.get_test_operations():
        print(f" {op.code:4d}: {op.name}")

    print("\n== REPAIR OPERATIONS ==")
    for op in api.process.get_repair_operations():
        print(f" {op.code:4d}: {op.name}")

    print("\n== WIP OPERATIONS ==")
    for op in api.process.get_wip_operations():
        print(f" {op.code:4d}: {op.name}")
```

Pattern 3: Process Code Lookup Table

```
def build_process_lookup(api):
    """Build lookup table for process codes"""

    lookup = {
        "test": {},
        "repair": {},
        "wip": {}
    }

    for proc in api.process.get_processes():
        if proc.is_test_operation:
            lookup["test"][proc.code] = proc.name
        elif proc.is_repair_operation:
            lookup["repair"][proc.code] = proc.name
        elif proc.is_wip_operation:
            lookup["wip"][proc.code] = proc.name

    return lookup

# Usage
lookup = build_process_lookup(api)
print(f"Test operation 100: {lookup['test'].get(100, 'Unknown')}"")
```

Pattern 4: Automatic Process Selection

```
def get_appropriate_process(api, operation_type):
    """Get appropriate process code for operation type"""

    if operation_type == "test":
        return api.process.get_default_test_code()
    elif operation_type == "repair":
        return api.process.get_default_repair_code()
    else:
        wip_ops = api.process.get_wip_operations()
        return wip_ops[0].code if wip_ops else 200
```

Internal API (Advanced)

The process service provides additional internal API functionality:

```

# Get processes with full details (ProcessID, etc.)
processes = api.process.get_all_processes()

# Get repair operation configurations
configs = api.process.get_repair_operation_configs()

# Get repair categories (fail codes)
categories = api.process.get_repair_categories(500)

# Get flattened fail codes
fail_codes = api.process.get_fail_codes(500)

```

 **Warning:** These methods use internal API endpoints that may change without notice.
Check the docstrings for  INTERNAL API warnings.

Model Reference

ProcessInfo

Field	Type	Description
code	int	Process code (e.g., 100, 500)
name	str	Process name
description	str	Process description
is_test_operation	bool	True if test operation
is_repair_operation	bool	True if repair operation
is_wip_operation	bool	True if WIP operation
process_id	UUID	Process GUID (internal API)
process_index	int	Process order index
state	int	Process state (1=active)

RepairCategory (Internal API)

Field	Type	Description
guid	UUID	Category identifier
description	str	Category name
selectable	bool	Can be selected
sort_order	int	Display order
failure_type	int	Failure type code
fail_codes	List	Nested fail codes

RepairOperationConfig (Internal API)

Field	Type	Description
description	str	Configuration name
uut_required	int	UUT required flag
bom_required	int	BOM required flag
vendor_required	int	Vendor required flag
comp_ref_mask	str	Component reference regex
categories	List	Repair categories

Best Practices

1. Use Caching Effectively

```
# Don't disable caching unless necessary
api.process.refresh_interval = 300 # Keep reasonable interval

# Only refresh when needed
if need_fresh_data:
    api.process.refresh()
```

2. Validate Before Operations

```
# Always validate process codes
if not api.process.is_valid_test_operation(process_code):
    raise ValueError(f"Invalid process code: {process_code}")
```

3. Use Type-Specific Methods

```
# Good - type-safe lookup
test_op = api.process.get_test_operation(100)

# Less safe - could return wrong type
process = api.process.get_process(100)
```

4. Handle Missing Processes

```
# Always check for None
process = api.process.get_test_operation(code)
if process is None:
    # Use default or raise error
    code = api.process.get_default_test_code()
```

Troubleshooting

Process Not Found

```
# Check if process exists
process = api.process.get_process(code)
if process is None:
    # List all available processes
    print("Available processes:")
    for p in api.process.get_processes():
        print(f"  {p.code}: {p.name}")
```

Wrong Process Type

```
# get_test_operation returns None if not a test operation
process = api.process.get_test_operation(500) # Returns None (500 is repair)

# Use generic get_process to check type
process = api.process.get_process(500)
if process:
    if process.is_repair_operation:
        print("This is a repair operation, not a test operation")
```

Stale Cache Data

```
# Force refresh if you expect new processes
api.process.refresh()

# Or reduce refresh interval
api.process.refresh_interval = 60 # 1 minute
```

Limitations

Read-Only Operations

Process definitions are **read-only** through the PyWATS API.

Creating, updating, or deleting processes is not supported.

Process management must be done through the WATS Admin interface:

- Add new test operations
- Configure repair operations
- Set up WIP operations

The API provides:

- List all processes
- Filter by process type
- Look up by code or name
- Validate process codes
- Create processes
- Update processes
- Delete processes

Related Documentation

- Report Module - Test reports using process codes

- Production Module - Production with process tracking
 - Software Module - Software distribution
-

Source: [docs/usage/product-module.md](#)

Product Module Usage Guide

Overview

The Product module manages products (parts), revisions, BOMs (Bill of Materials), and box build templates in WATS.

Quick Start

```
from pywats import pyWATS

api = pyWATS(base_url="https://wats.example.com", token="credentials")

# Get all products
products = api.product.get_products()

# Get specific product
product = api.product.get_product("PART-001")

# Get revisions
revisions = api.product.get_revisions("PART-001")
```

Basic Operations

1. Get Products

```
# Get all products (summary view)
products = api.product.get_products()

for product in products:
    print(f"{product.part_number}: {product.description}")
    print(f"  State: {product.state}")

# Get product with full details
product = api.product.get_product_full("PART-001")
print(f"Created: {product.created_date}")
print(f"Revisions: {len(product.revisions)}")
```

2. Get Specific Product

```
from pywats import ProductState

# Get by part number
product = api.product.get_product("MODULE-100")

if product:
    print(f"Part Number: {product.part_number}")
    print(f"Description: {product.description}")
    print(f"State: {product.state}")
    print(f"Active: {product.state == ProductState.ACTIVE}")
```

3. Create Product

```
from pywats import ProductState

# Create new product
product = api.product.create_product(
    part_number="NEW-PART-001",
    description="New Product Description",
    state=ProductState.ACTIVE,
    product_group="Electronics" # Optional
)

print(f"Created product: {product.part_number}")
```

4. Update Product

```
# Get product, modify, update
product = api.product.get_product("PART-001")
product.description = "Updated Description"
product.state = ProductState.ACTIVE

updated = api.product.update_product(product)
```

Product Revisions

1. Get Revisions

```
# Get all revisions for a product
revisions = api.product.get_revisions("PART-001")

for rev in revisions:
    print(f"Revision {rev.revision}: {rev.state}")

# Get specific revision
revision = api.product.get_revision("PART-001", "B")
```

2. Create Revision

```
from pywats import ProductState

# Create new revision
revision = api.product.create_revision(
    part_number="PART-001",
    revision="C",
    state=ProductState.ACTIVE,
    description="Revision C changes"
)
```

3. Revision States

```
from pywats.domains.product.enums import ProductState

# Available states
ProductState.ACTIVE      # Active/in production
ProductState.Obsolete     # No longer used
ProductState.Development  # In development
ProductState.Prototype    # Prototype phase
```

Product Tags

Products and revisions can have tags (key-value metadata):

1. Product Tags

```
# Get product tags
tags = api.product.get_product_tags("PART-001")
for tag in tags:
    print(f"{tag.key}: {tag.value}")

# Set product tags (replaces all)
api.product.set_product_tags("PART-001", [
    {"key": "Category", "value": "Power Supply"},
    {"key": "Voltage", "value": "12V"}
])

# Add single tag
api.product.add_product_tag("PART-001", "Manufacturer", "ACME Corp")
```

2. Revision Tags

```
# Get revision tags
tags = api.product.get_revision_tags("PART-001", "A")

# Set revision tags
api.product.set_revision_tags("PART-001", "A", [
    {"key": "PCB_Version", "value": "1.2"},
    {"key": "Test_Program", "value": "v3.1"}
])
```

Bill of Materials (BOM)

1. Get BOM

```
# Get BOM for a revision
bom = api.product.get_bom("ASSEMBLY-001", "A")

for item in bom.items:
    print(f"{item.part_number} - Qty: {item.quantity}")
    print(f"  Designators: {item.designators}")
    print(f"  Revision Mask: {item.revision_mask}")
```

2. Upload BOM

```
# Upload BOM from list
bom_items = [
    {
        "part_number": "RESISTOR-100",
        "quantity": 10,
        "designators": "R1,R2,R3,R4,R5,R6,R7,R8,R9,R10",
        "revision_mask": "*" # Any revision
    },
    {
        "part_number": "CAPACITOR-10UF",
        "quantity": 5,
        "designators": "C1,C2,C3,C4,C5",
        "revision_mask": "A|B" # Revision A or B only
    }
]

api.product.upload_bom(
    part_number="ASSEMBLY-001",
    revision="A",
    bom_items=bom_items
)
```

3. Revision Masks

Revision masks control which component revisions are acceptable:

```
# Common patterns:
"*"      # Any revision
"A"      # Only revision A
"A|B"    # Revision A or B
"A|B|C"  # Revision A, B, or C
"[A-Z]"  # Any letter A through Z
```

4. Upload BOM from Dict

```
# Alternative: Upload from dictionary
bom_dict = {
    "RESISTOR-100": {"quantity": 10, "designators": "R1-R10", "revision_mask": "*"},
    "CAPACITOR-10UF": {"quantity": 5, "designators": "C1-C5", "revision_mask": "*"}
}

api.product.upload_bom_from_dict("ASSEMBLY-001", "A", bom_dict)
```

Box Build Templates (Internal API)

Box build templates define assemblies with sub-units (e.g., PCBAs in modules).

 **Note:** This uses internal WATS APIs that may change.

1. Get Box Build Template

```
# Get template showing required subunits
template = api.product.get_box_build_template("MODULE-100", "A")

print(f"Main part: {template.part_number}")
print("Required subunits:")
for subunit in template.subunits:
    print(f"  - {subunit.part_number} (Index: {subunit.index})")
    print(f"    Revision mask: {subunit.revision_mask}")
```

2. Get Required Parts

```
# Get list of parts needed for assembly
parts = api.product.get_box_build_subunits("MODULE-100", "A")

for part in parts:
    print(f"{part.part_number} - {part.description}")
```

3. Add Subunit to Box Build

```
# Get template, add subunit, and save
template = api.product.get_box_build_template("MODULE-100", "A")
template.add_subunit(
    child_part="PCBA-200",
    child_revision="A",
    revision_mask="*", # Accept any revision
    index=0 # Position in assembly
)
template.save()
```

4. Remove Subunit

```
# Get template, remove subunit, and save
template = api.product.get_box_build_template("MODULE-100", "A")
template.remove_subunit("PCBA-200", "A") # Remove by part and revision
template.save()
```

5. Box Build Context Manager

```
# Convenient way to define box build
with api.product.get_box_build_template("MODULE-100", "A") as builder:
    builder.add_subunit("PCBA-200", "A")
    builder.add_subunit("PCBA-201", "A")
    # Auto-saved when context exits
```

Product Groups

1. Get Product Groups

```
# Get all product groups
groups = api.product.get_product_groups()

for group in groups:
    print(group)
```

2. Filter by Group

```
# Get all products in a group
products = api.product.get_products()
electronics = [p for p in products if p.product_group == "Electronics"]
```

Common Patterns

Pattern 1: Product Setup Workflow

```
from pywats import ProductState

# 1. Create product
product = api.product.create_product(
    part_number="NEW-MODULE",
    description="New Power Module",
    state=ProductState.DEVELOPMENT
)

# 2. Create initial revision
revision = api.product.create_revision(
    part_number="NEW-MODULE",
    revision="A",
    state=ProductState.DEVELOPMENT
)

# 3. Add tags
api.product.set_product_tags("NEW-MODULE", [
    {"key": "Type", "value": "Power Supply"}, 
    {"key": "Voltage", "value": "12V"}
])

# 4. Upload BOM
bom = [...] # Your BOM items
api.product.upload_bom("NEW-MODULE", "A", bom)

# 5. Activate when ready
revision.state = ProductState.ACTIVE
api.product.update_revision(revision)
```

Pattern 2: Revision Upgrade

```
# Copy settings from old to new revision
old_rev = api.product.get_revision("PART-001", "A")
old_bom = api.product.get_bom("PART-001", "A")
old_tags = api.product.get_revision_tags("PART-001", "A")

# Create new revision
new_rev = api.product.create_revision(
    part_number="PART-001",
    revision="B",
    state=ProductState.DEVELOPMENT,
    description=f"Based on Rev {old_rev.revision}"
)

# Copy BOM (modify as needed)
api.product.upload_bom("PART-001", "B", old_bom.items)

# Copy tags
api.product.set_revision_tags("PART-001", "B", old_tags)

# When tested, activate new and obsolete old
new_rev.state = ProductState.ACTIVE
old_rev.state = ProductState.OBSOLETE
api.product.update_revision(new_rev)
api.product.update_revision(old_rev)
```

Pattern 3: Assembly Definition

```
# Define a module with subassemblies
MAIN_MODULE = "MODULE-500W"
MAIN_REV = "A"

# Create main product
api.product.create_product(MAIN_MODULE, "500W Power Module", ProductState.ACTIVE)
api.product.create_revision(MAIN_MODULE, MAIN_REV, ProductState.ACTIVE)

# Define subunits
with api.product.get_box_build_template(MAIN_MODULE, MAIN_REV) as builder:
    builder.add_subunit("PCBA-CONTROL", "A")
    builder.add_subunit("PCBA-POWER", "A")
    builder.add_subunit("FAN-ASSEMBLY", "A")

# Upload BOM for mechanical parts
mechanical_bom = [
    {"part_number": "SCREW-M3", "quantity": 8, "designators": "S1-S8", "revision_mask": "*"},
    {"part_number": "HEATSINK", "quantity": 1, "designators": "HS1", "revision_mask": "*"}
]
api.product.upload_bom(MAIN_MODULE, MAIN_REV, mechanical_bom)
```

Pattern 4: BOM Validation

```
def validate_bom(part_number: str, revision: str):
    """Validate BOM has all required parts"""
    bom = api.product.get_bom(part_number, revision)

    issues = []
    for item in bom.items:
        # Check if referenced part exists
        part = api.product.get_product(item.part_number)
        if not part:
            issues.append(f"Part not found: {item.part_number}")
            continue

        # Check if part has active revision matching mask
        revisions = api.product.get_revisions(item.part_number)
        active_revs = [r for r in revisions if r.state == ProductState.ACTIVE]

        if not active_revs:
            issues.append(f"No active revision for: {item.part_number}")

    return issues
```

Best Practices

1. Use Meaningful Part Numbers

```
# ✓ Good - descriptive
"MODULE-500W-12V"
"PCBA-CONTROLLER-V2"
"RESISTOR-10K-0603"

# X Avoid - unclear
"PART001"
"ASM-1"
```

2. Manage States Properly

```
# ✓ Good - proper lifecycle
product.state = ProductState.DEVELOPMENT # During design
product.state = ProductState.ACTIVE      # In production
product.state = ProductState.OBSOLETE    # End of life

# X Avoid - skipping states
product.state = ProductState.ACTIVE # Directly from creation (risky)
```

3. Use Revision Masks

```
# ✓ Good - flexible BOM
{"part_number": "RESISTOR", "revision_mask": "*"} # Any revision OK

# ✓ Good - controlled BOM
{"part_number": "FIRMWARE", "revision_mask": "3.0|3.1"} # Specific versions

# X Avoid - too restrictive
{"part_number": "RESISTOR", "revision_mask": "A"} # Must update for every revision
```

4. Tag Consistently

```
# ✓ Good - consistent naming
api.product.add_product_tag("PART-001", "Category", "Electronics")
api.product.add_product_tag("PART-002", "Category", "Mechanical")

# X Avoid - inconsistent
api.product.add_product_tag("PART-001", "category", "electronics") # Lowercase
api.product.add_product_tag("PART-002", "Type", "Mechanical") # Different key
```

5. Validate Before Production

```
# ✓ Good - validate before activating
product = api.product.get_product("NEW-PART")
bom = api.product.get_bom("NEW-PART", "A")

if len(bom.items) > 0: # Has BOM
    revision = api.product.get_revision("NEW-PART", "A")
    revision.state = ProductState.ACTIVE
    api.product.update_revision(revision)
else:
    print("ERROR: Cannot activate - BOM is empty")
```

Troubleshooting

Product Not Found

```
# Check if product exists before operations
product = api.product.get_product("PART-001")
if not product:
    print("Product doesn't exist - create it first")
    product = api.product.create_product("PART-001", ...)
```

BOM Upload Failing

```
# Ensure all referenced parts exist
for item in bom_items:
    part = api.product.get_product(item["part_number"])
    if not part:
        print(f"Create {item['part_number']} first")
```

Revision Conflicts

```
# Check existing revisions before creating new
revisions = api.product.get_revisions("PART-001")
existing = [r.revision for r in revisions]
if "B" in existing:
    print("Revision B already exists")
```

Related Documentation

- Production Module - For manufacturing units from products
- Report Module - For testing products
- Architecture - Overall system design

Source: docs/usage/production-module.md

Production Module Usage Guide

Overview

The Production module manages serial numbers, unit tracking, production phases, and assembly operations in WATS.

Quick Start

```
from pywats import pyWATS

api = pyWATS(base_url="https://wats.example.com", token="credentials")

# Get unit information
unit = api.production.get_unit("SN-12345")

# Verify unit status
verification = api.production.verify_unit("SN-12345")
print(f"Passing: {verification.is_passing}")

# Check if unit is passing
is_passing = api.production.is_unit_passing("SN-12345")
```

Serial Number Management

1. Allocate Serial Numbers

```
# Get available serial number types for a product
sn_types = api.production.get_serial_number_types("PART-001", "A")

for sn_type in sn_types:
    print(f"{sn_type.name}: {sn_type.pattern}")

# Allocate serial numbers (if using WATS allocation)
serial_numbers = api.production.allocate_serial_numbers(
    part_number="PART-001",
    revision="A",
    quantity=10,
    serial_type_id=1 # From get_serial_number_types
)

for sn in serial_numbers:
    print(f"Allocated: {sn}")
```

2. Manual Serial Numbers

```
# For customer-provided or pre-existing serials
serial_number = "CUSTOMER-SN-001"

# Create unit with this serial (happens automatically on first test/operation)
# No explicit allocation needed
```

Unit Operations

1. Get Unit Information

```
# Get full unit details
unit = api.production.get_unit("SN-12345")

if unit:
    print(f"Serial: {unit.serial_number}")
    print(f"Part: {unit.part_number} Rev {unit.revision}")
    print(f"State: {unit.state}")
    print(f"Phase: {unit.phase}")
    print(f"Current Process: {unit.current_process}")
    print(f"Last Test: {unit.last_test_date}")
```

2. Create/Update Units

```
from pywats.domains.production import Unit

# Units are typically created automatically when first tested
# But you can create them explicitly:

unit = Unit(
    serial_number="SN-NEW-001",
    part_number="PART-001",
    revision="A"
)

created_unit = api.production.create_unit(unit)
```

3. Set Unit Phase

```
# Common phases:
# - "Undefined"
# - "Under Production - Queued"
# - "Under Production"
# - "Finalized"
# - "Scrapped"

# Set unit to production queue
api.production.set_unit_phase("SN-12345", "Under Production - Queued")

# Start production
api.production.set_unit_phase("SN-12345", "Under Production")

# Finalize after passing all tests
api.production.set_unit_phase("SN-12345", "Finalized")
```

4. Set Unit Process

```
# Set current operation/process
# Process codes come from app.get_processes()

api.production.set_unit_process(
    serial_number="SN-12345",
    operation_type=100 # ICT, Final Test, etc.
)
```

5. Update Unit Tags

```
# Add or update tags (metadata) on units
api.production.set_unit_tags("SN-12345", [
    {"key": "LotNumber", "value": "LOT-2025-W01"},
    {"key": "WorkOrder", "value": "WO-12345"},
    {"key": "Station", "value": "ICT-3"}
])
```

Unit Verification

1. Verify Unit Status

```
from pywats.domains.production.enums import UnitVerificationGrade

# Get verification with grade
verification = api.production.verify_unit("SN-12345")

print(f"Grade: {verification.grade}")
print(f"Passing: {verification.is_passing}")
print(f"Failed Steps: {verification.failed_step_count}")

# Check specific grades
if verification.grade == UnitVerificationGrade.PASSED:
    print("All tests passed")
elif verification.grade == UnitVerificationGrade.FAILED:
    print("Some tests failed")
elif verification.grade == UnitVerificationGrade.NOT_TESTED:
    print("Not yet tested")
```

2. Simple Pass/Fail Check

```
# Quick check - returns boolean
if api.production.is_unit_passing("SN-12345"):
    print("Unit is passing")
    api.production.set_unit_phase("SN-12345", "Finalized")
else:
    print("Unit has failures")
    # Send to repair or scrap
```

3. Verification Grades

```
from pywats.domains.production.enums import UnitVerificationGrade

# Available grades:
UnitVerificationGrade.PASSED          # All tests passed
UnitVerificationGrade.FAILED           # Has failures
UnitVerificationGrade.NOT_TESTED      # No test data
UnitVerificationGrade.PENDING          # Tests in progress
```

Unit Changes (History)

1. Get Change History

```
# Get all changes for a unit
changes = api.production.get_unit_changes("SN-12345")

for change in changes:
    print(f"{change.timestamp}: {change.change_type}")
    print(f"  Process: {change.process_name}")
    print(f"  Operator: {change.operator}")
    print(f"  Phase: {change.new_phase}")
```

2. Track Unit Lifecycle

```
def print_unit_lifecycle(serial_number: str):
    """Print complete unit lifecycle"""
    changes = api.production.get_unit_changes(serial_number)

    print(f"\nLifecycle for {serial_number}:")
    print("=" * 60)

    for change in sorted(changes, key=lambda c: c.timestamp):
        print(f"{change.timestamp:%Y-%m-%d %H:%M:%S}")
        print(f"  Type: {change.change_type}")
        print(f"  Process: {change.process_name or 'N/A'}")
        print(f"  Phase: {change.old_phase} → {change.new_phase}")
        print()
```

Assembly Operations

1. Verify Assembly (Box Build)

```
# Check if all required subunits are present
verification = api.production.verify_assembly(
    parent_serial="MODULE-SN-001",
    parent_part="MODULE-100",
    parent_revision="A"
)

print(f"Complete: {verification.is_complete}")
print(f"Missing parts: {verification.missing_count}")

for missing in verification.missing_parts:
    print(f"  - {missing.part_number} at index {missing.index}")
```

2. Build Assembly

```
# Add subunit to assembly (using production service)
api.production.add_child_to_assembly(
    parent_serial="MODULE-SN-001",
    child_serial="PCBA-SN-123",
    index=0 # Position in assembly (from box build template)
)

# Add another subunit
api.production.add_child_to_assembly(
    parent_serial="MODULE-SN-001",
    child_serial="PCBA-SN-124",
    index=1
)

# Verify assembly is complete
verification = api.production.verify_assembly("MODULE-SN-001", "MODULE-100", "A")
if verification.is_complete:
    print("Assembly complete!")
```

3. Disassemble

```
# Remove subunit from assembly
api.production.remove_child_from_assembly(
    parent_serial="MODULE-SN-001",
    index=0 # Position to remove
)
```

Production Batches

1. Track Batch/Lot

```
# Group units by batch using tags
batch_number = "BATCH-2025-W01"

# Set batch on units as they're created
for serial in serial_numbers:
    api.production.set_unit_tags(serial, [
        {"key": "LotNumber", "value": batch_number},
        {"key": "ManufactureDate", "value": "2025-01-15"}
    ])
```

2. Query Batch Units

```
# Get all units in batch (via report query with OData)
# Note: Tag-based filtering may require specific OData syntax
headers = api.report.query_uut_headers(
    odata_filter="partNumber eq 'WIDGET-001'",
    top=500
)

# Filter by tag in application code
batch_units = [
    r for r in headers
    if any(t.get("key") == "LotNumber" and t.get("value") == "BATCH-2025-W01"
        for t in getattr(r, 'misc_info', [])) or []
]

print(f"Batch has {len(batch_units)} units")
```

Common Patterns

Pattern 1: Complete Production Workflow

```
def production_workflow(serial_number: str, part_number: str, revision: str):
    """Complete workflow from creation to finalization"""

    # 1. Set to production queue
    api.production.set_unit_phase(serial_number, "Under Production - Queued")
    api.production.set_unit_tags(serial_number, [
        {"key": "LotNumber", "value": "LOT-2025-W01"},
        {"key": "StartDate", "value": datetime.now().isoformat()}
    ])

    # 2. Start production - ICT
    api.production.set_unit_phase(serial_number, "Under Production")
    api.production.set_unit_process(serial_number, operation_type=10) # ICT

    # Run ICT test (creates UUT report)
    # ... test code ...

    # 3. Move to next operation - Final Test
    api.production.set_unit_process(serial_number, operation_type=50)

    # Run final test
    # ... test code ...

    # 4. Verify and finalize
    if api.production.is_unit_passing(serial_number):
        api.production.set_unit_phase(serial_number, "Finalized")
        print(f"{serial_number}: Production complete")
    else:
        print(f"{serial_number}: Failed - send to repair")
```

Pattern 2: Assembly Build Workflow

```
def build_module(module_sn: str, pcba_serials: list):
    """Build module from PCBAs"""

    MODULE_PART = "MODULE-100"
    MODULE_REV = "A"

    # 1. Create module unit
    api.production.set_unit_phase(module_sn, "Under Production - Queued")

    # 2. Verify all PCBAs are passing
    for pcba_sn in pcba_serials:
        if not api.production.is_unit_passing(pcba_sn):
            raise ValueError(f"PCBA {pcba_sn} is not passing")

    # 3. Build assembly
    for index, pcba_sn in enumerate(pcba_serials):
        api.production.add_child_to_assembly(
            parent_serial=module_sn,
            child_serial=pcba_sn,
            index=index
        )

    # 4. Verify assembly complete
    verification = api.production.verify_assembly(module_sn, MODULE_PART, MODULE_REV)
    if not verification.is_complete:
        raise ValueError(f"Assembly incomplete: {verification.missing_parts}")

    # 5. Test module
    api.production.set_unit_phase(module_sn, "Under Production")
    api.production.set_unit_process(module_sn, operation_type=100)

    # ... run module tests ...

    # 6. Finalize if passing
    if api.production.is_unit_passing(module_sn):
        api.production.set_unit_phase(module_sn, "Finalized")
```

Pattern 3: Repair Workflow

```
def repair_workflow(serial_number: str):
    """Handle failed unit repair"""

    # 1. Check current status
    verification = api.production.verify_unit(serial_number)

    if verification.is_passing:
        print("Unit is already passing")
        return

    # 2. Create UUR (repair) report
    from pywats.models import UURReport

    # Get last test to determine failure
    unit = api.production.get_unit(serial_number)

    # Create repair report
    uur = api.report.create_uur_from_part_and_process(
        part_number=unit.part_number,
        serial_number=serial_number,
        revision=unit.revision,
        process_code=unit.current_process,
        failure_category=500, # Your failure category
        failure_code=501,      # Specific failure
        description="Repair performed",
        operator="Repair Tech"
    )

    api.report.send_uur_report(uur)

    # 3. Retest
    api.production.set_unit_process(serial_number, operation_type=unit.current_process)

    # ... run test again ...

    # 4. Check if now passing
    if api.production.is_unit_passing(serial_number):
        print(f"{serial_number}: Repair successful")
    else:
        print(f"{serial_number}: Still failing - escalate")
```

Pattern 4: Production Dashboard

```
def production_dashboard(part_number: str):
    """Show production status for a product"""
    from datetime import datetime, timedelta

    # Get all reports for product using OData filter
    start_date = datetime.now() - timedelta(days=7)

    headers = api.report.query_uut_headers(
        odata_filter=f"partNumber eq '{part_number}' and start ge {start_date.strftime('%Y-%m-%d')}",
        top=1000
    )

    # Analyze
    total = len(headers)
    passed = len([h for h in headers if h.passed])
    failed = total - passed

    print(f"\n{part_number} Production Status (Last 7 Days)")
    print("=" * 60)
    print(f"Total Tested: {total}")
    print(f"Passed: {passed} ({passed/total*100:.1f}%)")
    print(f"Failed: {failed} ({failed/total*100:.1f}%)")

    # Show units in production
    in_production = []
    for header in headers:
        unit = api.production.get_unit(header.serial_number)
        if unit and unit.phase == "Under Production":
            in_production.append(unit.serial_number)

    print(f"\nCurrently in production: {len(in_production)}")
    for sn in in_production[:10]: # Show first 10
        print(f" - {sn}")
```

Best Practices

1. Use Phases Consistently

```
# ✓ Good - proper flow
api.production.set_unit_phase(sn, "Under Production - Queued") # 1. Queued
api.production.set_unit_phase(sn, "Under Production")           # 2. Testing
api.production.set_unit_phase(sn, "Finalized")                  # 3. Done

# X Avoid - skipping phases
api.production.set_unit_phase(sn, "Finalized") # Directly (missing history)
```

2. Verify Before Finalizing

```
# ✓ Good - always check
if api.production.is_unit_passing(sn):
    api.production.set_unit_phase(sn, "Finalized")
else:
    # Handle failure

# ✗ Avoid - finalizing without checking
api.production.set_unit_phase(sn, "Finalized") # What if it failed?
```

3. Track with Tags

```
# ✓ Good - comprehensive tracking
api.production.set_unit_tags(sn, [
    {"key": "LotNumber", "value": "LOT-001"}, 
    {"key": "WorkOrder", "value": "WO-12345"}, 
    {"key": "Station", "value": "ICT-3"}, 
    {"key": "Operator", "value": "John Doe"}
])

# ✗ Avoid - minimal tracking
# (Hard to trace issues later)
```

4. Validate Assembly Order

```
# ✓ Good - build in order
for index, pcba_sn in enumerate(pcba_serials):
    api.production.add_child_to_assembly(module_sn, pcba_sn, index)

# ✗ Avoid - wrong indices
api.production.add_child_to_assembly(module_sn, pcba1, 1) # Should be 0
api.production.add_child_to_assembly(module_sn, pcba2, 0) # Should be 1
```

Troubleshooting

Unit Not Found

```
# Units are created on first test or explicit creation
unit = api.production.get_unit("NEW-SN-001")
if not unit:
    print("Unit will be created on first test")
```

Assembly Verification Failing

```
# Check template definition
template = api.product.get_box_build_template("MODULE-100", "A")
print(f"Required subunits: {len(template.subunits)}")

# Verify each subunit
verification = api.production.verify_assembly("MODULE-SN-001", "MODULE-100", "A")
for missing in verification.missing_parts:
    print(f"Missing: {missing.part_number} at index {missing.index}")
```

Process Code Not Found

```
# Get available process codes
processes = api.app.get_processes()
for proc in processes:
    print(f"{proc.code}: {proc.name}")
```

Related Documentation

- Report Module - For creating test reports
- Product Module - For product/BOM setup
- Architecture - Overall system design

Source: docs/usage/rootcause-module.md

RootCause Module Usage Guide

Overview

The RootCause module provides a ticketing system for tracking quality issues, root cause investigations, and corrective actions in WATS. It supports the 8D (Eight Disciplines) problem-solving methodology commonly used in electronics manufacturing.

Quick Start

```
from pywats import pyWATS
from pywats.domains.rootcause import TicketStatus, TicketPriority, TicketView

api = pyWATS(base_url="https://wats.example.com", token="credentials")

# Get open tickets assigned to you
tickets = api.rootcause.get_tickets(
    status=TicketStatus.OPEN,
    view=TicketView.ASSIGNED
)

# Create a new ticket
ticket = api.rootcause.create_ticket(
    subject="Solder Bridge Defect - ICT Line 1",
    priority=TicketPriority.HIGH,
    initial_comment="Detected 5% defect rate on PCBA-001"
)
```

Ticket Status

```
from pywats.domains.rootcause import TicketStatus

# Individual statuses (IntFlag - can be combined)
TicketStatus.OPEN      # 1 - New ticket
TicketStatus.IN_PROGRESS # 2 - Being worked on
TicketStatus.ON_HOLD    # 4 - Temporarily paused
TicketStatus.SOLVED     # 8 - Solution found
TicketStatus.CLOSED     # 16 - Closed
TicketStatus.ARCHIVED   # 32 - Archived

# Combining statuses for filtering
active = TicketStatus.OPEN | TicketStatus.IN_PROGRESS # Open or In Progress
all_active = TicketStatus.OPEN | TicketStatus.IN_PROGRESS | TicketStatus.ON_HOLD
```

Ticket Priority

```
from pywats.domains.rootcause import TicketPriority

TicketPriority.LOW      # 0 - Low priority
TicketPriority.MEDIUM   # 1 - Normal priority
TicketPriority.HIGH     # 2 - High priority (production impact)
```

Ticket Views

```
from pywats.domains.rootcause import TicketView

TicketView.ASSIGNED    # 0 - Tickets assigned to you
TicketView.FOLLOWING   # 1 - Tickets you're following
TicketView.ALL          # 2 - All tickets (requires permission)
```

Basic Operations

1. Get Tickets

```
# Get tickets assigned to you (default view)
my_tickets = api.rootcause.get_tickets(
    status=TicketStatus.OPEN | TicketStatus.IN_PROGRESS,
    view=TicketView.ASSIGNED
)

for ticket in my_tickets:
    print(f"#{{ticket.ticket_number}}: {{ticket.subject}}")
    print(f"  Status: {{TicketStatus(ticket.status).name}}")
    print(f"  Priority: {{TicketPriority(ticket.priority).name}}")
    print(f"  Assignee: {{ticket.assignee}}")

# Get all open tickets
all_open = api.rootcause.get_tickets(
    status=TicketStatus.OPEN,
    view=TicketView.ALL
)

# Get active tickets (convenience method)
active = api.rootcause.get_active_tickets()  # OPEN | IN_PROGRESS
```

2. Get Single Ticket

```
# Get ticket by ID
ticket = api.rootcause.get_ticket(ticket_id)

if ticket:
    print(f"Ticket #{ticket.ticket_number}")
    print(f"Subject: {ticket.subject}")
    print(f"Owner: {ticket.owner}")
    print(f"Assignee: {ticket.assignee}")
    print(f"Team: {ticket.team}")
    print(f"Created: {ticket.created_utc}")

# History (comments and updates)
if ticket.history:
    print(f"History entries: {len(ticket.history)}")
    for update in ticket.history:
        print(f" - {update.update_utc}: {update.content[:50]}...")
```

3. Create Ticket

```
# Basic ticket
ticket = api.rootcause.create_ticket(
    subject="ICT Failure Rate Spike",
    priority=TicketPriority.HIGH,
    initial_comment="Failure rate increased from 1% to 5% this morning."
)

# Ticket linked to a test report
ticket = api.rootcause.create_ticket(
    subject=f"Failure Investigation - {serial_number}",
    priority=TicketPriority.HIGH,
    report_uuid=report_id, # Link to failing report
    initial_comment="Investigating test failure on unit..."
)

# Ticket with assignee and team
ticket = api.rootcause.create_ticket(
    subject="Process Issue - Reflow Profile",
    priority=TicketPriority.MEDIUM,
    assignee="quality_engineer",
    team="Quality Team",
    initial_comment="Temperature profile may need adjustment."
)
```

4. Update Ticket

```
# Get ticket, modify, update
ticket = api.rootcause.get_ticket(ticket_id)
ticket.priority = TicketPriority.HIGH
ticket.team = "Engineering Team"

updated = api.rootcause.update_ticket(ticket)
```

5. Add Comment

```
# Add comment to ticket
api.rootcause.add_comment(
    ticket_id=ticket.ticket_id,
    comment="Completed 5-Why analysis. Root cause identified as incorrect stencil."
)
```

6. Change Status

```
# Change ticket status
api.rootcause.change_status(
    ticket_id=ticket.ticket_id,
    status=TicketStatus.IN_PROGRESS
)

# Mark as solved
api.rootcause.change_status(
    ticket_id=ticket.ticket_id,
    status=TicketStatus.SOLVED
)
```

7. Assign Ticket

```
# Assign to user
api.rootcause.assign_ticket(
    ticket_id=ticket.ticket_id,
    assignee="john.smith"
)
```

8. Archive Tickets

```
# Archive solved tickets
api.rootcause.archive_tickets([ticket_id1, ticket_id2])

# Note: Only SOLVED tickets can be archived
```

Team Assignment with Tags

Tags can be used to store team member roles:

```
from pywats.shared import Setting

# Get ticket and add team member tags
ticket = api.rootcause.get_ticket(ticket_id)

team_tags = [
    Setting(key="Team_Champion", value="J. Smith"),
    Setting(key="Team_Leader", value="M. Johnson"),
    Setting(key="Team_ProcessEng", value="K. Williams"),
    Setting(key="Team_TestEng", value="L. Chen"),
    Setting(key="Team_QualityEng", value="R. Patel"),
]

# Add tags to ticket
if ticket.tags:
    ticket.tags.extend(team_tags)
else:
    ticket.tags = team_tags

api.rootcause.update_ticket(ticket)
```

8D Problem-Solving Workflow

The 8D methodology is a structured approach to problem solving. Here's how to implement it with RootCause tickets:

D0: Preparation

```
# Create ticket for the problem
ticket = api.rootcause.create_ticket(
    subject=f"[8D] Solder Bridge Defect - {serial_number}",
    priority=TicketPriority.HIGH,
    report_uuid=failing_report_id,
    initial_comment=""""
## D0: Preparation

### Emergency Response Actions
1. Quarantine affected batch
2. Stop production on affected line
3. Notify Quality Manager

### Symptom Description
- Defect: Solder bridge on U3 (MCU)
- Impact: ICT test failure
- Urgency: HIGH - Production stopped
"""
)
```

D1: Establish Team

```
# Document team formation
d1_comment = """
## D1: Team Assembly

### Cross-Functional Team

| Role | Name | Responsibility |
|-----|-----|-----|
| Champion | J. Smith | Executive sponsor |
| Leader | M. Johnson | Investigation lead |
| Process Engineer | K. Williams | Manufacturing process |
| Test Engineer | L. Chen | Test system expert |
| Quality Engineer | R. Patel | Quality documentation |

### Team Charter
- Objective: Identify root cause and implement corrective action
- Timeline: 2 weeks
- Meetings: Daily standup at 09:00
"""

api.rootcause.add_comment(ticket.ticket_id, d1_comment)
api.rootcause.change_status(ticket.ticket_id, TicketStatus.IN_PROGRESS)
```

D2: Problem Definition

```
# Document problem using 5W2H
d2_comment = """
## D2: Problem Definition (5W2H)

### WHAT is the problem?
Solder bridges on U3 MCU pins 12-13

### WHERE was it found?
ICT Station 01, Production Line 2

### WHEN did it occur?
Started 2024-12-13 morning shift

### WHO found it?
Automated ICT test system

### WHY is it a problem?
- Causes test failures
- 5% defect rate (threshold: 1%)
- Production line stopped

### HOW MANY affected?
5 units out of 100 tested

### HOW was it detected?
Boundary scan test during ICT
"""

api.rootcause.add_comment(ticket.ticket_id, d2_comment)
```

D3: Interim Containment

```
d3_comment = """
## D3: Interim Containment Actions

### Actions Taken

| Action | Owner | Status |
|-----|-----|-----|
| Quarantine batch L2024-1213 | Technician | Complete |
| 100% visual inspection | QC | Complete |
| Increase SPI sampling to 100% | Process Eng | Complete |
| Add manual inspection post-reflow | Supervisor | Complete |

### Verification

- 50 units quarantined and inspected
- 5 total defective units identified
- 45 good units released
- No escapes to customer
"""

api.rootcause.add_comment(ticket.ticket_id, d3_comment)
```

D4: Root Cause Analysis

```
d4_comment = """
## D4: Root Cause Analysis

### 5-Why Analysis

1. **Why?** Too much solder paste on pads
2. **Why?** Stencil aperture oversized
3. **Why?** Wrong stencil revision used (Rev A vs Rev B)
4. **Why?** Stencil not verified before changeover
5. **Why?** No stencil verification in changeover procedure

**ROOT CAUSE:** Missing stencil verification step in changeover procedure

### Verification

- Test: Used correct stencil on 20 units
- Result: 0 defects
- Conclusion: Root cause confirmed
"""

api.rootcause.add_comment(ticket.ticket_id, d4_comment)
```

D5-D8: Continue Pattern

Continue documenting each phase with comments until closure.

Common Patterns

Pattern 1: Create Ticket from Test Failure

```
def create_failure_ticket(report_id, serial_number, failure_description):
    """Create RootCause ticket from a test failure"""

    ticket = api.rootcause.create_ticket(
        subject=f"Test Failure - {serial_number}",
        priority=TicketPriority.HIGH,
        report_uuid=report_id,
        initial_comment=f"""
## Test Failure Report

**Serial Number:** {serial_number}
**Report ID:** {report_id}

### Failure Description
{failure_description}

### Initial Assessment
Pending investigation.

---
*Auto-generated from test system*
""")

    return ticket
```

Pattern 2: Get My Active Tickets

```
def get_my_active_tickets():
    """Get all active tickets assigned to current user"""

    tickets = api.rootcause.get_tickets(
        status=TicketStatus.OPEN | TicketStatus.IN_PROGRESS | TicketStatus.ON_HOLD,
        view=TicketView.ASSIGNED
    )

    # Sort by priority (HIGH first)
    tickets.sort(key=lambda t: t.priority if t.priority else 0, reverse=True)

    return tickets
```

Pattern 3: Ticket Summary Report

```
def generate_ticket_summary():
    """Generate summary of all tickets by status"""

    all_tickets = api.rootcause.get_tickets(
        status=TicketStatus.OPEN | TicketStatus.IN_PROGRESS | TicketStatus.ON_HOLD | TicketStatus.SOLVED,
        view=TicketView.ALL
    )

    # Group by status
    by_status = {}
    for ticket in all_tickets:
        status = TicketStatus(ticket.status).name if ticket.status else "UNKNOWN"
        if status not in by_status:
            by_status[status] = []
        by_status[status].append(ticket)

    # Print summary
    print("=" * 50)
    print("ROOTCAUSE TICKET SUMMARY")
    print("=" * 50)

    for status, tickets in by_status.items():
        print(f"\n{status}: {len(tickets)} tickets")
        for t in tickets[:5]: # Show first 5
            priority = TicketPriority(t.priority).name if t.priority else "?"
            print(f"  #{t.ticket_number} [{priority}] {t.subject[:40]}...")
```

Pattern 4: Close and Archive Workflow

```
def close_ticket(ticket_id, resolution_comment):
    """Close a ticket with resolution comment"""

    # Add resolution comment
    api.rootcause.add_comment(ticket_id, f"""
## Resolution

{resolution_comment}

---
*Ticket closed by automated workflow*
""")

    # Change to SOLVED
    api.rootcause.change_status(ticket_id, TicketStatus.SOLVED)

    return api.rootcause.get_ticket(ticket_id)

def archive_old_solved_tickets(days_old=30):
    """Archive tickets solved more than X days ago"""
    from datetime import datetime, timedelta

    solved = api.rootcause.get_tickets(
        status=TicketStatus.SOLVED,
        view=TicketView.ALL
    )

    cutoff = datetime.now() - timedelta(days=days_old)
    to_archive = []

    for ticket in solved:
        if ticket.updated_utc and ticket.updated_utc < cutoff:
            to_archive.append(ticket.ticket_id)

    if to_archive:
        api.rootcause.archive_tickets(to_archive)
        print(f"Archived {len(to_archive)} tickets")
```

Ticket Model Reference

Ticket Fields

Field	Type	Description
ticket_id	UUID	Unique identifier
ticket_number	int	Human-readable ticket number
subject	str	Ticket title
status	TicketStatus	Current status
priority	TicketPriority	Priority level
owner	str	Ticket owner username
assignee	str	Assigned user
team	str	Assigned team
report_uuid	UUID	Linked test report ID
created_utc	datetime	Creation timestamp
updated_utc	datetime	Last update timestamp
progress	str	Progress notes
origin	str	Origin/source
tags	List[Setting]	Key-value tags
history	List[TicketUpdate]	Update history

TicketUpdate Fields

Field	Type	Description
update_id	UUID	Update identifier
update_utc	datetime	Update timestamp
update_user	str	User who made update
content	str	Update content
update_type	TicketUpdateType	Type of update
attachments	List[TicketAttachment]	Attached files

Best Practices

1. Use Clear Subjects

```
# Good - descriptive subjects
"[8D] Solder Bridge on U3 - Batch L2024-1213"
"ICT Failure Spike - Line 2 Morning Shift"
"Customer Return - RMA-2024-001"

# Avoid - vague subjects
"Problem"
"Issue"
"Check this"
```

2. Link to Reports

```
# Always link tickets to related test reports
ticket = api.rootcause.create_ticket(
    subject="...",
    report_uuid=report_id, # Link to failing report
    ...
)
```

3. Document Progress

```
# Add comments at each stage
api.rootcause.add_comment(ticket_id, "Started 5-Why analysis...")
api.rootcause.add_comment(ticket_id, "Root cause identified: ...")
api.rootcause.add_comment(ticket_id, "Corrective action implemented: ...")
```

4. Use Proper Status Transitions

```
# Typical flow:
# OPEN -> IN_PROGRESS -> SOLVED -> ARCHIVED

# Or with holds:
# OPEN -> IN_PROGRESS -> ON_HOLD -> IN_PROGRESS -> SOLVED
```

5. Assign Appropriately

```
# Assign to specific user for accountability
api.rootcause.assign_ticket(ticket_id, "quality_engineer")

# Use team for group ownership
ticket.team = "Quality Team"
api.rootcause.update_ticket(ticket)
```

Troubleshooting

Ticket Not Found

```
ticket = api.rootcause.get_ticket(ticket_id)
if not ticket:
    # Try searching
    tickets = api.rootcause.get_tickets(
        status=TicketStatus.OPEN | TicketStatus.IN_PROGRESS | TicketStatus.SOLVED,
        view=TicketView.ALL
    )
    # Search by subject or other criteria
```

Cannot Archive

```
# Only SOLVED tickets can be archived
ticket = api.rootcause.get_ticket(ticket_id)
if ticket.status != TicketStatus.SOLVED:
    api.rootcause.change_status(ticket_id, TicketStatus.SOLVED)

api.rootcause.archive_tickets([ticket_id])
```

View Returns Empty

```
# ASSIGNED view requires tickets assigned to current user
# Use ALL view to see everything (requires permission)
tickets = api.rootcause.get_tickets(
    status=TicketStatus.OPEN,
    view=TicketView.ALL # Instead of ASSIGNED
)
```

Related Documentation

- Product Module - Products involved in issues
- Production Module - Production units with failures
- Report Module - Test reports linked to tickets

Source: docs/usage/software-module.md

Software Module Usage Guide

Overview

The Software module provides functionality for managing software distribution packages in WATS. These packages are used to distribute software, configurations, and files to test stations and production equipment.

Quick Start

```
from pywats import pyWATS
from pywats.domains.software import PackageStatus

api = pyWATS(base_url="https://wats.example.com", token="credentials")

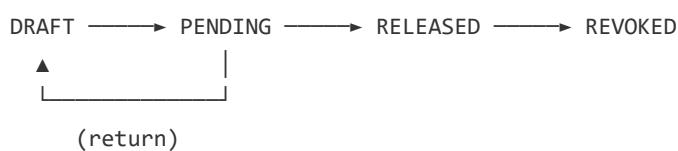
# Get all software packages
packages = api.software.get_packages()

# Get a released package by name
package = api.software.get_released_package("TestExecutive")

# Create a new package
new_pkg = api.software.create_package(
    name="MyTestSoftware",
    description="Test station software v2.0",
    priority=10
)
```

Package Status Workflow

Software packages follow a strict status workflow:



Status	Description	Can Edit	Can Delete
DRAFT	Initial state, under development	All fields	<input checked="" type="checkbox"/> Yes
PENDING	Submitted for review	Status, Tags only	<input type="checkbox"/> No
RELEASED	Approved for distribution	Status, Tags only	<input type="checkbox"/> No
REVOKED	Withdrawn from distribution	-	<input checked="" type="checkbox"/> Yes

```
from pywats.domains.software import PackageStatus

# Status enum values
PackageStatus.DRAFT      # "Draft"
PackageStatus.PENDING     # "Pending"
PackageStatus.RELEASED   # "Released"
PackageStatus.REVOKED    # "Revoked"
```

Basic Operations

1. List All Packages

```
# Get all packages
packages = api.software.get_packages()

for pkg in packages:
    print(f"{pkg.name} v{pkg.version} [{pkg.status.value}]")
```

2. Get Package by ID

```
# Get specific package
package = api.software.get_package(package_id)

if package:
    print(f"Name: {package.name}")
    print(f"Version: {package.version}")
    print(f"Status: {package.status.value}")
    print(f"Description: {package.description}")
```

3. Get Package by Name

```
# Get package by name and status
package = api.software.get_package_by_name(
    name="TestExecutive",
    status=PackageStatus.RELEASED,
    version=2 # Optional - omit for latest
)

# Convenience method for released packages
released = api.software.get_released_package("TestExecutive")
```

4. Create Package

```
from pywats.domains.software import PackageTag

# Create with basic info
package = api.software.create_package(
    name="MyTestSoftware",
    description="Automated test software for production",
    install_on_root=False,
    root_directory="/TestStation",
    priority=10,
)

# Create with tags
tags = [
    PackageTag(key="platform", value="windows"),
    PackageTag(key="category", value="test-executive"),
    PackageTag(key="version", value="2.0.0"),
]

package = api.software.create_package(
    name="TaggedSoftware",
    description="Software with metadata tags",
    tags=tags,
)
```

Note: If a package with the same name exists, the new version will be incremented automatically.

5. Update Package

```
# Get package
package = api.software.get_package(package_id)

# Update metadata (only in DRAFT status)
package.description = "Updated description"
package.priority = 20

updated = api.software.update_package(package)
```

Note: Only DRAFT packages allow full editing. PENDING and RELEASED packages can only have Status and Tags updated.

6. Delete Package

```
# Delete by ID (must be DRAFT or REVOKED)
success = api.software.delete_package(package_id)

# Delete by name and version
success = api.software.delete_package_by_name("MyTestSoftware", version=1)
```

Status Workflow Methods

Submit for Review (Draft → Pending)

```
success = api.software.submit_for_review(package_id)
if success:
    print("Package submitted for review")
```

Return to Draft (Pending → Draft)

```
success = api.software.return_to_draft(package_id)
if success:
    print("Package returned to draft")
```

Release Package (Pending → Released)

```
success = api.software.release_package(package_id)
if success:
    print("Package released for distribution")
```

Revoke Package (Released → Revoked)

```
success = api.software.revoke_package(package_id)
if success:
    print("Package revoked")
```

Package Files

List Package Files

```
files = api.software.get_package_files(package_id)

for f in files:
    size_mb = f.size / (1024 * 1024) if f.size else 0
    print(f"{f.filename} - {size_mb:.2f} MB")
    print(f"  Path: {f.path}")
    print(f"  Checksum: {f.checksum}")
    print(f"  Attributes: {f.attributes}")
```

Upload Zip File

```
# Read zip file
with open("software_package.zip", "rb") as f:
    zip_content = f.read()

# Upload to package (merge with existing)
success = api.software.upload_zip(
    package_id=package_id,
    zip_content=zip_content,
    clean_install=False # Merge files
)

# Upload with clean install (delete existing first)
success = api.software.upload_zip(
    package_id=package_id,
    zip_content=zip_content,
    clean_install=True # Delete all existing files first
)
```

Zip File Requirements:

- Files cannot be at the root level of the zip
- All files must be in a folder: zipFile/myFolder/myFile.txt

Update File Attributes

```
# Get files first
files = api.software.get_package_files(package_id)

# Update attribute for a file
target_file = next(f for f in files if f.filename == "setup.exe")

success = api.software.update_file_attribute(
    file_id=target_file.file_id,
    attributes="ExecuteOnce"
)
```

File Attribute Values:

Attribute Description	
----- -----	
None No special handling	
ExecuteOnce Execute once after install	
ExecuteAlways Execute on every sync	
TopLevelFile Display in package root	
OverwriteNever Never overwrite existing	
OverwriteOnNewPackageVersion Only overwrite on new version	
ExecuteOncePerVersion Execute once per package version	

Package Tags

Tags provide metadata for filtering and organizing packages.

Create Package with Tags

```
from pywats.domains.software import PackageTag

tags = [
    PackageTag(key="platform", value="windows"),
    PackageTag(key="environment", value="production"),
    PackageTag(key="owner", value="test-engineering"),
]

package = api.software.create_package(
    name="TaggedPackage",
    description="Package with tags",
    tags=tags,
)
```

Filter by Tag

```
# Get packages with specific tag
packages = api.software.get_packages_by_tag(
    tag="platform",
    value="windows",
    status=PackageStatus.RELEASED
)

for pkg in packages:
    print(f"{pkg.name} v{pkg.version}")
```

Virtual Folders

Virtual folders are registered directories in Production Manager.

```
# Get all virtual folders
folders = api.software.get_virtual_folders()

for folder in folders:
    print(f"{folder.name}: {folder.path}")
    print(f"  Description: {folder.description}")
```

Common Patterns

Pattern 1: Create and Release Package

```
def create_and_release_package(api, name, description, zip_path):
    """Create a package and release it for distribution"""

    # 1. Create draft package
    package = api.software.create_package(
        name=name,
        description=description,
    )

    if not package:
        raise Exception("Failed to create package")

    # 2. Upload files
    with open(zip_path, "rb") as f:
        zip_content = f.read()

    api.software.upload_zip(
        package_id=package.package_id,
        zip_content=zip_content,
        clean_install=True
    )

    # 3. Submit for review
    api.software.submit_for_review(package.package_id)

    # 4. Release (requires approval workflow in production)
    api.software.release_package(package.package_id)

    return api.software.get_package(package.package_id)
```

Pattern 2: Get Latest Released Package

```
def get_latest_package(api, name):
    """Get the latest released version of a package"""
    return api.software.get_package_by_name(
        name=name,
        status=PackageStatus.RELEASED
        # Omit version to get highest version
    )
```

Pattern 3: Package Version Comparison

```
def is_newer_version_available(api, name, current_version):
    """Check if a newer released version exists"""
    latest = api.software.get_released_package(name)

    if latest and latest.version:
        return latest.version > current_version
    return False
```

Pattern 4: Revoke and Replace Package

```
def replace_package(api, old_pkg_id, new_zip_path):
    """Revoke old package and create replacement"""

    # Get old package info
    old_pkg = api.software.get_package(old_pkg_id)

    # Revoke old package (if released)
    if old_pkg.status == PackageStatus.RELEASED:
        api.software.revoke_package(old_pkg_id)

    # Create new version (auto-increments version number)
    new_pkg = api.software.create_package(
        name=old_pkg.name,
        description=old_pkg.description,
        priority=old_pkg.priority,
        tags=old_pkg.tags,
    )

    # Upload new files
    with open(new_zip_path, "rb") as f:
        api.software.upload_zip(new_pkg.package_id, f.read(), clean_install=True)

    return new_pkg
```

Model Reference

Package

Field	Type	Description
package_id	UUID	Unique identifier
name	str	Package name
description	str	Package description
version	int	Version number (auto-incremented)
status	PackageStatus	Current status
install_on_root	bool	Install at root level
root_directory	str	Root installation directory
priority	int	Installation priority
tags	List[PackageTag]	Metadata tags
created_utc	datetime	Creation timestamp
modified_utc	datetime	Last modification
created_by	str	Creator username
modified_by	str	Last modifier username
files	List[PackageFile]	Package files (when populated)

PackageFile

Field	Type	Description
file_id	UUID	File identifier
filename	str	File name
path	str	Full path within package
size	int	File size in bytes
checksum	str	File checksum
attributes	str	File attributes
created_utc	datetime	Creation timestamp
modified_utc	datetime	Last modification

PackageTag

Field	Type	Description
key	str	Tag name/key
value	str	Tag value

VirtualFolder

Field	Type	Description
folder_id	UUID	Folder identifier
name	str	Folder name
path	str	Folder path
description	str	Folder description

Best Practices

1. Use Meaningful Names

```
# Good - descriptive name with version info  
"TestExecutive_EOL_v2"  
"Calibration_Tools_ICT"  
  
# Avoid - vague names  
"Software1"  
"Package"
```

2. Use Tags for Organization

```
# Organize by metadata  
tags = [  
    PackageTag(key="platform", value="windows"),  
    PackageTag(key="station_type", value="ICT"),  
    PackageTag(key="maintainer", value="test-team"),  
    PackageTag(key="semantic_version", value="2.1.0"),  
]
```

3. Always Test Before Release

```
# Create and test in DRAFT  
package = api.software.create_package(...)  
api.software.upload_zip(package.package_id, zip_content)  
  
# Verify files  
files = api.software.get_package_files(package.package_id)  
assert len(files) > 0  
  
# Then release  
api.software.submit_for_review(package.package_id)  
api.software.release_package(package.package_id)
```

4. Clean Up Test Packages

```
# Delete test packages after testing  
if package.status in [PackageStatus.DRAFT, PackageStatus.REVOKED]:  
    api.software.delete_package(package.package_id)
```

Troubleshooting

Cannot Delete Package

```
# Must be DRAFT or REVOKED to delete
package = api.software.get_package(package_id)

if package.status == PackageStatus.RELEASED:
    api.software.revoke_package(package_id)
elif package.status == PackageStatus.PENDING:
    api.software.return_to_draft(package_id)

# Now can delete
api.software.delete_package(package_id)
```

Zip Upload Fails

```
# Ensure zip has proper structure
# Files must be in a folder, not at root
# Correct: mypackage.zip/installer/setup.exe
# Wrong:   mypackage.zip/setup.exe (at root)
```

Status Transition Fails

```
# Only valid transitions:
# Draft -> Pending (submit_for_review)
# Pending -> Draft (return_to_draft)
# Pending -> Released (release_package)
# Released -> Revoked (revoke_package)

# Cannot skip steps (e.g., Draft -> Released)
```

Limitations

Package File Download Not Supported

Downloading package files is not currently supported through the PyWATS API.
Files can only be uploaded to packages, not downloaded.

To access package files, use the WATS web interface or Production Manager.

Related Documentation

- Process Module - Test/repair operations
 - Production Module - Production units
 - Asset Module - Test station assets
-

Source: [docs/usage/box-build-guide.md](#)

Box Build Guide

Overview

Box build functionality in pyWATS manages multi-level product assemblies where a parent product contains one or more child products (subunits). This guide explains how to:

1. Define what subunits a product requires (Box Build Template)
2. Actually build assemblies during production (Unit Assembly)

Key Concept: Templates vs. Units

Understanding the distinction between these two concepts is crucial:

Box Build Template (Product Domain)

What it is: A DESIGN-TIME definition of what subunits are required to build a product.

Where it lives: Product domain (`api.product`)

Example use case: "A Controller Module (CTRL-100) requires 1x Power Supply and 2x Sensor Board"

```
# This defines the REQUIREMENTS (what's needed)
template = api.product.get_box_build_template("CTRL-100", "A")
template.add_subunit("PSU-200", "A", quantity=1)
template.add_subunit("SENSOR-300", "A", quantity=2)
template.save()
```

Unit Assembly (Production Domain)

What it is: RUNTIME attachment of actual production units (with serial numbers) to a parent unit.

Where it lives: Production domain (api.production)

Example use case: "Unit CTRL-SN-001 now contains PSU-SN-456, SENSOR-SN-789, and SENSOR-SN-790"

```
# This BUILDS the assembly (attaches actual units)
api.production.add_child_to_assembly(
    parent_serial="CTRL-SN-001", parent_part="CTRL-100",
    child_serial="PSU-SN-456", child_part="PSU-200"
)
```

Complete Workflow

Step 1: Create Products

First, create the parent and child products with revisions:

```
from pywats import pyWATS
from pywats.domains.product.enums import ProductState

api = pyWATS(base_url="https://wats.example.com", token="your-token")

# Create child product (Power Supply)
api.product.create_product(
    part_number="PSU-200",
    name="Power Supply Unit",
    description="24V DC Power Supply",
    state=ProductState.ACTIVE
)
api.product.create_revision(
    part_number="PSU-200",
    revision="A",
    state=ProductState.ACTIVE
)

# Create parent product (Controller Module)
api.product.create_product(
    part_number="CTRL-100",
    name="Controller Module",
    description="Main Controller with Power Supply",
    state=ProductState.ACTIVE
)
api.product.create_revision(
    part_number="CTRL-100",
    revision="A",
    state=ProductState.ACTIVE
)
```

Step 2: Define Box Build Template

Define what subunits the parent product requires:

```
# Get or create box build template
template = api.product.get_box_build_template("CTRL-100", "A")

# Add required subunits
template.add_subunit(
    part_number="PSU-200",
    revision="A",
    quantity=1
)

# Save to server
template.save()

# Or use context manager (auto-saves)
with api.product.get_box_build_template("CTRL-100", "A") as bb:
    bb.add_subunit("PSU-200", "A", quantity=1)
    bb.add_subunit("SENSOR-300", "A", quantity=2)
# Automatically saved when exiting context
```

Step 3: Create Production Units

Create actual units with serial numbers:

```
from pywats.domains.production import Unit

# Create parent unit
parent_unit = Unit(
    serial_number="CTRL-SN-001",
    part_number="CTRL-100",
    revision="A"
)

# Create child unit
child_unit = Unit(
    serial_number="PSU-SN-456",
    part_number="PSU-200",
    revision="A"
)

# Save to WATS
api.production.create_units([parent_unit, child_unit])
```

Step 4: Test and Finalize Child Units

Important: Child units MUST be finalized before they can be added to an assembly.

```
# Run tests on child unit (creates test reports)
# ... your test code here ...

# Set child unit to "Finalized" phase
api.production.set_unit_phase(
    serial_number="PSU-SN-456",
    part_number="PSU-200",
    phase="Finalized" # or phase=16 (the phase ID)
)
```

Step 5: Build the Assembly

Attach child units to the parent:

```
# Add child to parent assembly
result = api.production.add_child_to_assembly(
    parent_serial="CTRL-SN-001",
    parent_part="CTRL-100",
    child_serial="PSU-SN-456",
    child_part="PSU-200"
)

if result:
    print("Child successfully added to assembly!")
```

Step 6: Verify Assembly

Check if all required subunits are attached:

```
# Verify assembly matches box build template
verification = api.production.verify_assembly(
    serial_number="CTRL-SN-001",
    part_number="CTRL-100",
    revision="A"
)

print(f"Verification result: {verification}")
```

API Reference

Product Domain (Templates)

```
api.product.get_box_build_template(part_number, revision)
```

Get or create a box build template.

```
template = api.product.get_box_build_template("CTRL-100", "A")
```

```
BoxBuildTemplate.add_subunit(part_number, revision, quantity=1)
```

Add a subunit requirement to the template.

```
template.add_subunit("PSU-200", "A", quantity=1)
template.add_subunit("SENSOR-300", "A", quantity=2, revision_mask="A,B")
```

Parameters:

- part_number : Child product part number
- revision : Default revision for the child
- quantity : How many are required (default: 1)
- revision_mask : Acceptable revisions pattern (optional)

```
BoxBuildTemplate.remove_subunit(part_number, revision)
```

Remove a subunit from the template.

```
template.remove_subunit("OLD-PART", "A")
```

```
BoxBuildTemplate.save()
```

Persist all changes to the server.

```
template.save()
```

```
BoxBuildTemplate.subunits
```

Get current subunits (including pending changes).

```
for subunit in template.subunits:
    print(f"{subunit.child_part_number} rev {subunit.child_revision}: qty {subunit.quantity}")
```

Production Domain (Unit Assembly)

```
api.production.add_child_to_assembly(parent_serial, parent_part, child_serial, child_part)
```

Attach a child unit to a parent assembly.

Requirements:

- Parent's box build template must define this child product

- Child unit must be in "Finalized" phase
- Child must not already have a parent

```
api.production.add_child_to_assembly(  
    parent_serial="CTRL-SN-001",  
    parent_part="CTRL-100",  
    child_serial="PSU-SN-456",  
    child_part="PSU-200"  
)
```

```
api.production.remove_child_from_assembly(parent_serial, parent_part, child_serial, child_part)
```

Detach a child unit from a parent.

```
api.production.remove_child_from_assembly(  
    parent_serial="CTRL-SN-001",  
    parent_part="CTRL-100",  
    child_serial="PSU-SN-456",  
    child_part="PSU-200"  
)
```

```
api.production.verify_assembly(serial_number, part_number, revision)
```

Check if assembly matches box build template.

```
result = api.production.verify_assembly("CTRL-SN-001", "CTRL-100", "A")
```

```
api.production.set_unit_phase(serial_number, part_number, phase)
```

Set a unit's phase. Use "Finalized" (or phase ID 16) before adding to assembly.

```
# By name  
api.production.set_unit_phase("PSU-SN-456", "PSU-200", "Finalized")  
  
# By ID  
api.production.set_unit_phase("PSU-SN-456", "PSU-200", 16)
```

Revision Masks

Revision masks control which child revisions are acceptable in a box build:

```
# Accept only revision A
template.add_subunit("PART", "A", revision_mask="A")

# Accept any revision starting with "1."
template.add_subunit("PART", "1.0", revision_mask="1.%")

# Accept multiple specific revisions
template.add_subunit("PART", "A", revision_mask="A,B,C")

# Accept revision range
template.add_subunit("PART", "1.0", revision_mask="1.0,1.1,1.2")
```

Error Handling

Common Errors

Child not finalized:

Error: Child unit must be in phase Finalized

Solution: Call api.production.set_unit_phase(child_sn, child_pn, "Finalized")

Child not in template:

Error: Parent's box build must define the child unit as valid

Solution: Add the child product to the box build template first

Child already has parent:

Error: The child unit must not already have a parent

Solution: Remove child from existing parent first

BOM vs Box Build

Bill of Materials (BOM): Lists electronic components (resistors, capacitors, ICs) - typically for PCB assembly.

Uses WSBF XML format.

Box Build Template: Lists subassemblies/subunits (PCBAs, power supplies, modules) - for mechanical/final assembly. Uses ProductRevisionRelation.

```

# BOM: Electronic components for PCB
bom_items = [
    BomItem(component_ref="R1", part_number="RES-10K", quantity=1),
    BomItem(component_ref="C1", part_number="CAP-100NF", quantity=1),
]
api.product.update_bom("PCBA-100", "A", bom_items)

# Box Build: Subassemblies for final product
template = api.product.get_box_build_template("MODULE-100", "A")
template.add_subunit("PCBA-100", "A", quantity=1) # The assembled PCB
template.add_subunit("PSU-200", "A", quantity=1) # Power supply
template.save()

```

Best Practices

- 1. Define templates before production:** Set up box build templates before creating production units.
- 2. Finalize children first:** Always finalize child units before adding to assembly.
- 3. Use revision masks wisely:** Allow flexibility where appropriate (e.g., "1.%" for minor revisions).
- 4. Verify after assembly:** Always call `verify_assembly()` to confirm completeness.
- 5. Handle errors gracefully:** Wrap assembly operations in try/except blocks.

```

try:
    api.production.add_child_to_assembly(
        parent_serial, parent_part, child_serial, child_part
    )
except Exception as e:
    print(f"Assembly failed: {e}")
    # Handle error (retry, log, alert, etc.)

```

Reference

Source: docs/env-variables.md

Environment Variables for pyWATS Client Debugging

Overview

For development and debugging, you can use environment variables to provide credentials without committing them to the repository.

Setup

1. Copy the example file:

```
bash cp .env.example .env
```

2. Edit .env with your credentials:

```
bash PYWATS_SERVER_URL=https://python.wats.com PYWATS_API_TOKEN=your_actual_token_here
```

3. Load environment variables:

PowerShell:

```
powershell # Load from .env file (requires dotenv package or manual loading) Get-Content .env |  
ForEach-Object { if ($_. -match '^([#=]+)=(.*)$') {  
[Environment]::SetEnvironmentVariable($matches[1], $matches[2], 'Process') } }
```

Bash/Linux:

```
bash export $(grep -v '^#' .env | xargs)
```

Or set directly:

```
powershell $env:PYWATS_SERVER_URL = "https://python.wats.com" $env:PYWATS_API_TOKEN =  
"cH1XQVRTX0FQ..."
```

1. Run the client:

```
bash python -m pywats_client service --instance-id default
```

How It Works

The `ClientConfig` class checks for environment variables at runtime via the `get_runtime_credentials()` method:

- `PYWATS_SERVER_URL` - Server URL (e.g., `https://python.wats.com`)
- `PYWATS_API_TOKEN` - API authentication token

Important Behavior:

1. **Environment variables are runtime-only** - They are NEVER saved to the config file
2. **Fallback mechanism** - Environment variables are only used if the config file has empty values
3. **Persistence** - Credentials entered through the GUI or saved to config.json are always persisted
4. **Development workflow** - Use env vars for debugging without modifying config files

This means:

- You can use env vars for debugging without them being saved to git
- Config file credentials (entered via GUI) persist between sessions
- Env vars only apply when config values are empty
- Changing credentials in GUI saves them properly to config.json

VS Code Integration

For VS Code debugging, add to `.vscode/launch.json`:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "pyWATS Client Service",
      "type": "python",
      "request": "launch",
      "module": "pywats_client",
      "args": ["service", "--instance-id", "default"],
      "env": {
        "PYWATS_SERVER_URL": "https://python.wats.com",
        "PYWATS_API_TOKEN": "your_token_here"
      }
    },
    {
      "name": "pyWATS Client GUI",
      "type": "python",
      "request": "launch",
      "module": "pywats_client",
      "args": ["gui", "--instance-id", "default"],
      "env": {
        "PYWATS_SERVER_URL": "https://python.wats.com",
        "PYWATS_API_TOKEN": "your_token_here"
      }
    }
  ]
}
```

Security Notes

- ✅ .env is in .gitignore - safe for local credentials
- ✅ Environment variables are runtime-only and NEVER saved to config files
- ✅ Config file credentials (from GUI) are properly persisted
- ✅ Env vars don't override saved config values
- ⚠️ **Never commit .env or tokens to git**
- ⚠️ **Use test tokens for development, not production tokens**

Getting Test Token

From test config:

```
# api-tests/instances/client_a_config.json
{
  "service_address": "https://python.wats.com",
  "api_token": "cH1XQVRTX0FQSV9BVVRPVEVTVD02cGhUUjg0ZTVIMHA1R3JUWgtQZ1Y0UTNvbmk2MiM="
}
```

Set in your local `.env` :

```
PYWATS_SERVER_URL=https://python.wats.com  
PYWATS_API_TOKEN=cH1XQVRTX0FQSV9BVVRPVEVTVD02cGhUUjg0ZTVIMHA1R3JUWgtQZ1Y0UTNvbmk2MiM=
```

Priority Order

1. **Config file values** (if not empty)
2. **Environment variables** (if config empty)
3. **User prompt** (if both empty)

This means you can have different configs per instance but use env vars as a fallback for debugging.

Source: docs/error-catalog.md

PyWATS Error Catalog

Last Updated: January 23, 2026

Version: 0.1.0b34

This document provides a comprehensive reference for all errors that can occur when using the pyWATS library. Each error includes its cause, examples, and remediation steps.

Table of Contents

- Error Handling Modes
- Exception Hierarchy
- Connection Errors
- Authentication & Authorization
- Data Validation Errors
- Resource Errors
- Server Errors
- Result Pattern Error Codes
- Client Errors
- Retry Behavior

- Quick Reference
-

Error Handling Modes

PyWATS supports two error handling modes that control how the library responds to ambiguous situations:

STRICT Mode (Default)

```
from pywats import pyWATS, ErrorMode

api = pyWATS(
    base_url="https://wats.example.com",
    token="your_token",
    error_mode=ErrorMode.STRICT  # Default
)
```

Behavior:

- Raises exceptions for all errors (404, validation, server errors)
- Raises `EmptyResponseError` when expecting data but receiving empty response
- Best for production code that needs certainty
- Fail-fast approach

LENIENT Mode

```
from pywats import pyWATS, ErrorMode

api = pyWATS(
    base_url="https://wats.example.com",
    token="your_token",
    error_mode=ErrorMode.LENIENT
)
```

Behavior:

- Returns `None` for 404 (resource not found)
- Returns `None` for empty responses
- Only raises exceptions for actual errors (validation, authentication, server errors)
- Best for exploratory code and scripts that handle missing data gracefully

Exception Hierarchy

```
Exception
└ PyWATSError (base for all pyWATS errors)
  └ WatsApiError (HTTP 4xx/5xx errors)
    └ AuthenticationError (401)
    └ AuthorizationError (403)
    └ ValidationError (400)
    └ ConflictError (409)
    └ ServerError (5xx)
  └ NotFoundError (404)
  └ EmptyResponseError (STRICT mode only)
  └ ConnectionError (network failures)
  └ TimeoutError (request timeouts)
```

All exceptions inherit from `PyWATSError`, so you can catch all pyWATS-specific errors with:

```
try:
    product = api.product.get_product("WIDGET-001")
except PyWATSError as e:
    print(f"PyWATS error: {e.message}")
    print(f"Operation: {e.operation}")
    print(f"Details: {e.details}")
```

Connection Errors

ConnectionError

HTTP Status: N/A (network level)

Error Code: CONNECTION_ERROR

Retry: Automatic (if retry enabled)

Cause:

- Cannot connect to WATS server
- DNS resolution failure
- Network unavailable
- Firewall blocking connection
- Server is down

Example:

```
from pywats import pyWATS, ConnectionError

api = pyWATS(base_url="https://unreachable-server.com", token="...")

try:
    api.test_connection()
except ConnectionError as e:
    print(f"Cannot connect: {e.message}")
    # Details: {'cause': 'Name or service not known'}
```

Remediation:

1. Verify server URL is correct
2. Check network connectivity (ping wats-server.com)
3. Verify DNS resolution (nslookup wats-server.com)
4. Check firewall/proxy settings
5. Confirm WATS server is running
6. Try accessing server URL in browser

Related Errors:

- TimeoutError - Connection attempt times out

TimeoutError

HTTP Status: N/A (network level)

Error Code: TIMEOUT

Retry: Automatic (if retry enabled)

Cause:

- Request takes longer than configured timeout
- Server is slow to respond
- Network latency issues
- Large data transfer

Example:

```
from pywats import pyWATS, TimeoutError

# Set a short timeout
api = pyWATS(base_url="...", token="...", timeout=5)

try:
    # Large query that takes >5 seconds
    reports = api.report.query_uut_headers(filter)
except TimeoutError as e:
    print(f"Request timed out: {e.message}")
```

Remediation:

1. Increase timeout value:

```
python api = pyWATS(base_url="...", token="...", timeout=60)
```
2. Narrow query scope (use filters, limit results)
3. Check network latency
4. Contact server administrator if persistent

Related Errors:

- ConnectionError - Cannot establish connection
-

Authentication & Authorization

AuthenticationError

HTTP Status: 401 Unauthorized

Error Code: UNAUTHORIZED

Retry: ✘ No (credentials need fixing)

Cause:

- Invalid credentials (username/password)
- Malformed authentication token
- Token not Base64 encoded properly
- Expired session

Example:

```
from pywats import pyWATS, AuthenticationError

api = pyWATS(base_url="...", token="invalid_token")

try:
    api.test_connection()
except AuthenticationError as e:
    print(f"Authentication failed: {e.message}")
    print(f"Status code: {e.status_code}") # 401
```

Remediation:

1. Verify credentials are correct
2. Ensure token is Base64 encoded:

```
python import base64
credentials = "username:password"
token = base64.b64encode(credentials.encode()).decode()
```
3. Check for special characters in password
4. Try authenticating through WATS web interface
5. Contact administrator to verify account status

Related Errors:

- AuthorizationError - Valid credentials but insufficient permissions
-

AuthorizationError

HTTP Status: 403 Forbidden

Error Code: FORBIDDEN

Retry: ✗ No (permissions issue)

Cause:

- User lacks required permissions for operation
- API endpoint requires elevated privileges
- Resource access restricted

Example:

```
from pywats import pyWATS, AuthorizationError

api = pyWATS(base_url="...", token="...")

try:
    # User doesn't have permission to create products
    product = api.product.create_product(...)
except AuthorizationError as e:
    print(f"Permission denied: {e.message}")
```

Remediation:

1. Verify user has required permissions in WATS
2. Contact administrator to grant necessary roles
3. Use account with appropriate access level
4. Check WATS user permissions in admin panel

Related Errors:

- AuthenticationError - Invalid credentials
-

Data Validation Errors

ReportHeaderValidationError

HTTP Status: N/A (client-side validation)

Error Code: PROBLEMATIC_CHARACTERS

Retry: ✗ No (fix input data or bypass intentionally)

Cause:

- Serial number or part number contains characters that cause issues with WATS searches/filters
- Problematic characters: * , % , ? , [,] , ^ , ! , / , \

Example:

```
from pywats.models import UUTReport
from pywats import ReportHeaderValidationError

try:
    report = UUTReport(
        pn="PART/001", # Contains '/' - problematic!
        sn="SN*001",   # Contains '*' - problematic!
        rev="A",
        process_code=10,
        station_name="TestStation",
        location="Lab",
        purpose="Development"
    )
except Exception as e: # Wrapped in pydantic ValidationError
    print(f"Problematic characters in report header")
```

Bypass Options:

When you intentionally need to use problematic characters:

Option 1: Context Manager

```
from pywats import allow_problematic_characters
from pywats.models import UUTReport

with allow_problematic_characters():
    report = UUTReport(
        pn="PART/001", # Now allowed (warning issued)
        sn="SN*001",   # Now allowed (warning issued)
        ...
    )
```

Option 2: SUPPRESS: Prefix

```
from pywats.models import UUTReport

report = UUTReport(
    pn="SUPPRESS:PART/001", # Prefix stripped, value = "PART/001"
    sn="SUPPRESS:SN*001",   # Prefix stripped, value = "SN*001"
    ...
)
```

Remediation:

1. Use only recommended characters: A-Z , a-z , 0-9 , - , _ , .
2. If problematic characters are required, use bypass options above
3. Be aware that bypassed values may cause issues with WATS searches/filters

Related Errors:

- `ValidationError` - General validation failures
-

ValidationError

HTTP Status: 400 Bad Request

Error Code: INVALID_INPUT , MISSING_REQUIRED_FIELD , INVALID_FORMAT , VALUE_OUT_OF_RANGE

Retry: ✘ No (fix input data)

Cause:

- Missing required fields
- Invalid data format
- Value out of acceptable range
- Failed Pydantic model validation
- Constraint violation

Example:

```
from pywats import pyWATS, ValidationError

api = pyWATS(base_url="...", token="...")

try:
    # Missing required field
    product = api.product.create_product(
        part_number="", # Empty part number
        state="ACTIVE"
    )
except ValidationError as e:
    print(f"Validation failed: {e.message}")
    print(f"Field: {e.field}") # 'part_number'
    print(f"Value: {e.value}") # ''
```

Common Scenarios:

Missing Required Field

```
# Error: part_number is required
api.product.create_product(description="Widget")

# Fix:
api.product.create_product(
    part_number="WIDGET-001",
    description="Widget"
)
```

Invalid Format

```
# Error: Invalid date format
filter = WATSFilter(start_date="2024-13-40") # Invalid date

# Fix:
from datetime import datetime
filter = WATSFilter(start_date=datetime(2024, 1, 15))
```

Pydantic Validation

```
from pywats.models import Product
from pydantic import ValidationError as PydanticValidationError

try:
    product = Product(
        part_number="WIDGET",
        state="INVALID_STATE" # Not a valid ProductState
    )
except PydanticValidationError as e:
    print(e.errors())
```

Remediation:

1. Check error message for specific field and issue
2. Verify all required fields are provided
3. Validate data types match expected types
4. Check value constraints (min/max, length, pattern)
5. Review model documentation for field requirements
6. Use Pydantic models for automatic validation

Related Errors:

- Pydantic ValidationError - Model validation failures

Resource Errors

NotFoundError

HTTP Status: 404 Not Found

Error Code: NOT_FOUND

Retry: ✗ No (resource doesn't exist)

Behavior:

- **STRICT mode:** Raises NotFoundError
- **LENIENT mode:** Returns None

Cause:

- Resource with given identifier doesn't exist
- Wrong part number, serial number, or ID
- Resource was deleted
- Typo in identifier

Example:

```
from pywats import pyWATS, NotFoundError

api = pyWATS(base_url="...", token="...", error_mode=ErrorMode.STRICT)

try:
    product = api.product.get_product("NONEXISTENT-PART")
except NotFoundError as e:
    print(f"Not found: {e.message}")
    print(f"Resource type: {e.resource_type}") # 'Product'
    print(f"Identifier: {e.identifier}") # 'NONEXISTENT-PART'
```

LENIENT mode example:

```
api = pyWATS(base_url="...", token="...", error_mode=ErrorMode.LENIENT)

product = api.product.get_product("MAYBE-EXISTS")
if product is None:
    print("Product not found, creating new one...")
else:
    print(f"Found: {product.part_number}")
```

Remediation:

1. Verify identifier is correct (check for typos)

2. List available resources to confirm existence:

```
python products = api.product.get_products() print([p.part_number for p in products])
```

3. Use search/query methods to find resource

4. Check if resource was deleted
5. Use LENIENT mode if `None` is acceptable

Related Errors:

- `EmptyResponseError` - Empty result set in STRICT mode
-

ConflictError

HTTP Status: 409 Conflict

Error Code: CONFLICT , ALREADY_EXISTS

Retry: ✘ No (conflict must be resolved)

Cause:

- Resource already exists (duplicate)
- Concurrent modification conflict
- Constraint violation (unique key)
- Version mismatch (optimistic locking)

Example:

```
from pywats import pyWATS, ConflictError

api = pyWATS(base_url="...", token="...")

try:
    # Part number already exists
    product = api.product.create_product(
        part_number="WIDGET-001", # Already exists
        description="Duplicate"
    )
except ConflictError as e:
    print(f"Conflict: {e.message}")
```

Remediation:

1. Check if resource already exists before creating
2. Use update instead of create for existing resources
3. Handle concurrent modifications with retry logic
4. Use unique identifiers
5. Implement optimistic locking if needed

Related Errors:

- `ValidationError` - Data validation failures
-

EmptyResponseError

HTTP Status: 200 OK (but body is empty)

Error Code: N/A

Retry: ✘ No

Mode: STRICT only

Cause:

- Server returned 200 OK but response body is empty/null
- Query returned no results
- Resource exists but has no data
- Server bug (should return 404)

Example:

```
from pywats import pyWATS, EmptyResponseError, ErrorMode

api = pyWATS(base_url="...", token="...", error_mode=ErrorMode.STRICT)

try:
    # Query returns empty result set
    reports = api.report.query_uut_headers(
        WATSFilter(part_number="NEVER-TESTED")
    )
except EmptyResponseError as e:
    print(f"Empty response: {e.message}")
    print(f"Operation: {e.operation}") # 'query_uut_headers'
```

Remediation:

1. Switch to LENIENT mode if empty results are acceptable:

```
python api = pyWATS(error_mode=ErrorMode.LENIENT) reports = api.report.query_uut_headers(filter)
if reports is None or len(reports) == 0: print("No reports found")
```

2. Broaden query filters

3. Verify resource should have data

4. Check if operation allows empty responses

Related Errors:

- NotFoundError - Resource doesn't exist (404)

Server Errors

ServerError

HTTP Status: 500-599 (Server errors)

Error Code: API_ERROR , OPERATION_FAILED

Retry: Automatic (if retry enabled)

Cause:

- Internal server error
- Database error
- Server bug
- Resource exhaustion
- Unhandled exception on server

Example:

```
from pywats import pyWATS, ServerError

api = pyWATS(base_url="...", token="...")

try:
    result = api.report.submit_report(report)
except ServerError as e:
    print(f"Server error: {e.message}")
    print(f"Status code: {e.status_code}") # 500, 502, 503, etc.
    print(f"Response body: {e.response_body}")
```

Common Status Codes:

- **500** - Internal Server Error
- **502** - Bad Gateway (proxy error)
- **503** - Service Unavailable (server overloaded)
- **504** - Gateway Timeout

Remediation:

1. Wait and retry (automatic retry is enabled by default)
2. Check WATS server logs for details
3. Contact server administrator
4. Reduce request frequency (see Retry Behavior)
5. Report bug to WATS support if persistent

Related Errors:

- `TimeoutError` - Request times out before server responds

Result Pattern Error Codes

When using the `Result[T]` pattern, methods return `Success[T]` or `Failure` instead of raising exceptions:

```
from pywats import pyWATS
from pywats.shared import Result, Success, Failure, ErrorCode

result: Result[Product] = api.product.some_result_based_method(...)

if result.is_success:
    product = result.value
    print(f"Success: {product.part_number}")
else:
    # result is Failure
    print(f"Error [{result.error_code}]: {result.message}")
    print(f"Details: {result.details}")
    print(f"Suggestions: {result.suggestions}")
```

Standard Error Codes

Error Code	Description	HTTP Equivalent
INVALID_INPUT	Input validation failed	400
MISSING_REQUIRED_FIELD	Required field not provided	400
INVALID_FORMAT	Data format is incorrect	400
VALUE_OUT_OF_RANGE	Value exceeds constraints	400
NOT_FOUND	Resource doesn't exist	404
ALREADY_EXISTS	Resource already exists	409
CONFLICT	Resource conflict	409
OPERATION_FAILED	Operation failed	500
SAVE_FAILED	Save operation failed	500
DELETE_FAILED	Delete operation failed	500
UNAUTHORIZED	Authentication required	401
FORBIDDEN	Insufficient permissions	403
CONNECTION_ERROR	Network failure	N/A
TIMEOUT	Request timeout	N/A
API_ERROR	Generic API error	500
UNKNOWN_ERROR	Unclassified error	N/A

Example Usage

```
from pywats.shared import Failure, ErrorCode

# Creating a failure
failure = Failure(
    error_code=ErrorCode.MISSING_REQUIRED_FIELD,
    message="part_number is required to create a product",
    details={"field": "part_number", "received": None},
    suggestions=["Provide a unique part_number string"]
)

# Handling failures
if result.is_failure:
    if result.error_code == ErrorCode.NOT_FOUND:
        print("Resource not found, creating new one...")
    elif result.error_code == ErrorCode.UNAUTHORIZED:
        print("Authentication required")
    else:
        print(f"Unexpected error: {result.message}")
```

Client Errors

These errors occur in the pyWATS Client application (`pywats_client`):

Configuration Errors

Cause:

- Invalid configuration file
- Missing required configuration
- Failed to load/save config

Example:

```
from pywats_client.core.config import ClientConfig

try:
    config = ClientConfig.load("invalid_config.json")
except Exception as e:
    print(f"Config error: {e}")
```

Remediation:

1. Verify config file exists and is valid JSON
2. Use `ClientConfig.load_or_create()` to create default if missing
3. Check file permissions
4. Validate required fields are present

Converter Errors

Cause:

- File format not recognized
- Invalid converter configuration
- Converter execution failure

Example:

```
from pywats_client.converters.models import ValidationResult

# Low confidence - file not recognized
result = converter.validate_file("unknown.dat")
if not result.can_convert:
    print(f"Cannot convert: {result.message}")
    print(f"Confidence: {result.confidence}") # < 0.5
```

Remediation:

1. Check file format matches converter expectations
2. Verify converter is enabled
3. Review converter configuration
4. Check converter logs for details
5. Ensure dependencies are met (ready=True)

Queue Errors

Cause:

- Failed to persist queue
- Queue corruption
- Disk space exhausted

Remediation:

1. Check disk space
2. Verify queue directory is writable
3. Clear corrupted queue files
4. Restart client

Retry Behavior

PyWATS automatically retries failed requests for transient errors:

Retryable Errors

The following errors trigger automatic retry:

- `ConnectionError` - Network failures
- `TimeoutError` - Request timeouts
- `ServerError` with status 429, 500, 502, 503, 504

Retry Configuration

```
from pywats import pyWATS, RetryConfig

# Default retry (3 attempts, exponential backoff)
api = pyWATS(base_url="...", token="...")

# Custom retry
config = RetryConfig(
    max_attempts=5,           # Try up to 5 times
    base_delay=2.0,            # Start with 2 second delay
    max_delay=60.0,             # Cap at 60 seconds
    exponential_base=2.0,       # Double delay each retry
    jitter=True,                # Add randomness to prevent thundering herd
)
api = pyWATS(base_url="...", token="...", retry_config=config)

# Disable retry
api = pyWATS(base_url="...", token="...", retry_enabled=False)
```

Retry Delay Calculation

```
delay = min(base_delay * (exponential_base ** attempt), max_delay)
if jitter:
    delay = delay * (0.5 + random.random() * 0.5)
```

Example delays (`base_delay=1.0, exponential_base=2.0`):

- Attempt 1: 1 second
- Attempt 2: 2 seconds
- Attempt 3: 4 seconds
- Attempt 4: 8 seconds
- Attempt 5: 16 seconds

Non-Retryable Errors

These errors are NOT retried (fix required):

- `AuthenticationError` (401)
- `AuthorizationError` (403)
- `ValidationError` (400)

- `NotFoundError` (404)
 - `ConflictError` (409)
-

Quick Reference

Common Error Patterns

```
from pywats import pyWATS, PyWATSError, NotFoundError, ValidationError, ConnectionError

api = pyWATS(base_url="...", token="...")

# Handle specific errors
try:
    product = api.product.get_product("WIDGET-001")
except NotFoundError:
    print("Product not found")
except ValidationError as e:
    print(f"Invalid input: {e.field} = {e.value}")
except ConnectionError:
    print("Cannot connect to server")
except PyWATSError as e:
    print(f"PyWATS error: {e.message}")

# Use LENIENT mode for optional resources
api = pyWATS(base_url="...", token="...", error_mode=ErrorMode.LENIENT)
product = api.product.get_product("MAYBE-EXISTS")
if product is None:
    print("Not found")
else:
    print(f"Found: {product.part_number}")

# Use Result pattern for structured errors
result = api.product.some_result_method(...)
if result.is_failure:
    print(f"[{result.error_code}] {result.message}")
    for suggestion in result.suggestions:
        print(f" - {suggestion}")
```

Debugging Tips

1. Enable debug logging:

```
python from pywats.core.logging import enable_debug_logging
enable_debug_logging()
```

2. Inspect exception details:

```
python try: api.product.get_product("X") except PyWATSError as e: print(f"Message:
{e.message}") print(f"Operation: {e.operation}") print(f"Details: {e.details}") if hasattr(e,
'cause') and e.cause: print(f"Cause: {e.cause}")
```

3. Check HTTP status codes:

```
python try: api.product.create_product(...) except WatsApiError as e: print(f"HTTP Status: {e.status_code}")
```

4. Use test_connection():

```
python if api.test_connection(): print("Connected successfully") else: print("Connection failed")
```

Getting Help

- **Documentation:** <https://github.com/olreppe/pyWATS/tree/main/docs>
- **GitHub Issues:** <https://github.com/olreppe/pyWATS/issues>
- **Email:** support@virinco.com
- **WATS Community:** <https://wats.com/community>

Error catalog version: 0.1.0b34

Last updated: January 23, 2026

Source: docs/wats-domain-knowledge.md

WATS Domain Knowledge for AI Agents

This document contains essential domain knowledge for AI agents working with WATS. Understanding these concepts is crucial for correctly interpreting and answering questions about manufacturing test data.

Core Concepts

Units, Reports, and Runs

Term	Definition
Unit	A single device/product being tested, identified by serial number
Report	The test result document from a single test execution
Run	A test execution sequence number (Run 1, Run 2, etc.)
Retest	Any run after Run 1

Key relationship: One unit can have multiple reports (one per run).

Process / Test Operation

A **process** (also called **test_operation**) is a type of test a product goes through:

- ICT (In-Circuit Test)
- FCT (Functional Test)
- EOL (End-of-Line Test)
- FQC (Final Quality Check)
- etc.

Critical: A product typically goes through MULTIPLE processes. Each process has its own yield metrics.

Operation Types (Terminology!)

WATS uses different operation types depending on the workflow:

Term	Domain	Records	Used For
test_operation	Report	UUT / UUTReport	Testing (yields, failures)
repair_operation	Report	UUR / UURReport	Repair logging (repair actions)
wip_operation	Production	WIP records	Production tracking only

Important: When users ask about "process" or "operation" for yield analysis, they almost always mean `test_operation`.

Only use `repair_operation` when specifically analyzing repair workflow (not yield).

Process Name Matching

The Problem

Users often use imprecise process names:

- "PCBA" instead of "PCBA test"
- "board test" instead of "PCBA Test Station"
- "fct" instead of "FCT Functional Test"

How to Handle

- 1. Fuzzy matching:** The agent tool attempts to match user input to actual process names
- 2. Common aliases:** Standard manufacturing abbreviations are recognized
- 3. Suggestions:** If no match found, suggest closest alternatives
- 4. List processes:** Use `perspective="by operation"` to see available processes

Common Aliases

User Input	May Match
"pcba", "board test"	PCBA test, PCBA Test, Board Test
"ict", "in-circuit"	ICT, ICT Test, In-Circuit Test
"fct", "functional"	FCT, FCT Test, Functional Test
"aoi", "optical"	AOI, AOI Test, Automated Optical Inspection
"eol", "end of line"	EOL, EOL Test, End of Line Test
"fqc", "final"	FQC, Final Quality Check

Best Practice

When in doubt, query available processes first:

```
# See all processes for a product
yield_tool.analyze(YieldFilter(
    part_number="WIDGET-001",
    perspective="by operation"
))
```

The Mixed Process Problem (Critical!)

The Scenario

Some customers send **different test types to the same process**:

Process "Structural Tests" receives:

- AOI tests (sw_filename: "aoi.exe")
- ICT tests (sw_filename: "ict.exe")

Why This Causes Problems

1. Unit's first run (AOI) determines that unit as "tested"
2. Unit's second run (ICT) is seen as a "retest after pass"
3. ICT yields show **0 units** because AOI already "passed" those units!

Symptoms

Symptom	Explanation
"ICT shows 0 units"	AOI ran first, ICT is treated as retest
"FPY doesn't make sense"	Mixed test types corrupt yield calculation
"Unit counts mismatch"	Different sw_filename = different "tests"

Diagnosis

Look for **different sw_filename values** in the same process:

- If multiple sw_filename exist for one process → Mixed process problem
- Each sw_filename represents a different test type

Agent Response

When user reports 0 units for a process that should have data:

1. **Check for mixed process:** Query reports, look for different sw_filename
 2. **Explain the issue:** "This process receives multiple test types (AOI, ICT). The first test type determines units."
 3. **Recommend solution:** "Each test type should have its own process for accurate yield tracking."
-

Adaptive Time Filtering

The Problem with 30-Day Default

For **high-volume production**, 30 days can be too much:

Volume	Daily Units	30 Days	Risk
Very High	>100,000	>3 million	API timeout
High	10K-100K	300K-3M	Slow, memory issues
Medium	1K-10K	30K-300K	OK
Low	<1K	<30K	May need more days

Adaptive Time Filter

The tool can **automatically adjust** the date range based on volume:

```
yield_tool.analyze(YieldFilter(  
    part_number="HIGH-VOLUME-PRODUCT",  
    adaptive_time=True  # Auto-adjust window  
)
```

How It Works

1. Start with small window (1 day)
2. Evaluate volume
3. Expand if insufficient data
4. Stop when target reached

When to Use

- High-volume environments
 - Unknown production volume
 - General queries
 - Specific date range needed
-

Yield Metrics (Critical Knowledge!)

The Process Context Rule

Every yield question must be considered in the context of a specific process.

Ambiguous: "What's the yield for WIDGET-001?"

Clear: "What's the FCT yield for WIDGET-001?"

Clear: "What's the RTY for WIDGET-001?" (overall across all processes)

Unit-Based Yield (FPY, SPY, TPY, LPY)

Metric	Name	Definition
FPY	First Pass Yield	% of units passed on Run 1
SPY	Second Pass Yield	% of units passed by Run 2
TPY	Third Pass Yield	% of units passed by Run 3
LPY	Last Pass Yield	% of units eventually passed

Relationship: FPY \leq SPY \leq TPY \leq LPY

Report-Based Yield (TRY)

Metric	Name	Definition
TRY	Test Report Yield	Passed reports \div All reports

Use TRY for: Station performance, fixture comparison, operator evaluation, repair line analysis.

Rolled Throughput Yield (RTY)

$$\text{RTY} = \text{FPY}_1 \times \text{FPY}_2 \times \dots \times \text{FPY}_n$$

RTY represents the probability that a unit passes ALL processes on the first try.

Example:

```
ICT FPY = 98%
FCT FPY = 95%
EOL FPY = 99%
```

$$\text{RTY} = 0.98 \times 0.95 \times 0.99 = 92.2\%$$

Use RTY for: Overall product quality assessment, comparing products across entire flow.

The Unit Inclusion Rule

A unit is included in yield calculations ONLY if its FIRST RUN matches the filter.

If included, ALL runs for that unit are counted (even runs outside the filter).

Implications:

- Filtering by retest-only stations shows 0 units
- Date filters apply to Run 1, not all runs
- Solution for retest stations: Use TRY instead of FPY

The Repair Line Problem

Repair/retest stations never see Run 1 (they only handle failed units from main line).

Result: Unit-based yield shows 0 units for repair stations.

Solution: Use TRY (report-based yield) to evaluate repair station performance.

Yield Over Time (Temporal Analysis)

Date Range Defaults

WATS always assumes you want the **most recent data**:

- If `date_from` and `date_to` are not specified, WATS defaults to **last 30 days**
- This is the server-side default behavior
- Use `days` parameter for simple "last N days" queries

Time-Based Perspectives

Perspective	Date Grouping	Use Case
trend	DAY	General yield trend
daily	DAY	Day-by-day breakdown
weekly	WEEK	Week-by-week analysis
monthly	MONTH	Month-by-month trends

Yield Trend Metrics

Yield trend describes **change compared to the previous equally-sized period**:

Analysis Period	Compared To
Today	Yesterday
This week	Last week
This month	Last month

Use Case: Detecting improvement or degradation patterns in production quality.

Safe Period Aggregation (Important!)

Key Rule: When fetching yield over periods, the **first-pass-included rule** applies.

This means:

- Units are counted only in their first-run period
- **Periods can be safely summed** without double-counting
- Example: Sum Monday-Friday yields for weekly total

Date Grouping Options

For advanced use, specify `date_grouping` directly:

Value	Groups By
HOUR	Hour
DAY	Day
WEEK	Week
MONTH	Month
QUARTER	Quarter
YEAR	Year

Top Runners

Definition

Top runners = Products with the highest test volume (unit count or report count).

The Per-Process Rule

Volume must be considered PER PROCESS.

A product might be a top runner in FCT but not in EOL:

Product	FCT Volume	EOL Volume
WIDGET-A	10,000 ★	5,000
WIDGET-B	8,000	12,000 ★

Finding Top Runners

```
# Top runners for FCT process
yield_tool.analyze(YieldFilter(
    test_operation="FCT",
    perspective="by product",
    days=30
))
# Results sorted by unit_count shows top runners
```

Handling Ambiguous Questions

Question: "What's the yield for WIDGET-001?"

Best response approach:

1. **Check if single process:** Query with `part_number="WIDGET-001"` and `perspective="by operation"`
2. If only ONE process: Answer with that process's yield
3. If MULTIPLE processes: Ask user to clarify
4. **Clarify with user:**
5. "WIDGET-001 goes through 3 processes (ICT, FCT, EOL). Which process yield would you like?"
6. "Or would you like the RTY (Rolled Throughput Yield) across all processes?"
7. **Default suggestion:** If user doesn't specify, show yield by operation so they can see all processes.

Question: "Show me the top runners"

Best response approach:

1. Ask for process context: "Top runners for which process? (e.g., FCT, EOL)"
 2. Or show overall volume by product across all processes as starting point
 3. Then drill down to specific process if needed
-

Unit Verification Rules

What They Are

Rules that define which processes must pass for each product before it can ship.

Example Rule

```
Product: WIDGET-001
Required tests: ICT → FCT → EOL
```

Common Issue

Many customers don't maintain verification rules, even though the API supports them.

Agent Opportunity

Analyze yield data to SUGGEST verification rules:

1. Query: part_number="WIDGET-001", perspective="by operation", days=90
 2. Identify all processes with significant volume
 3. Suggest: "Based on test data, WIDGET-001 should require: ICT → FCT → EOL"
 4. Offer to create the rule (if agent has edit permissions)
-

Quick Reference: When to Use Each Metric

Question Type	Recommended Metric
Product quality (single process)	FPY, LPY
Product quality (overall)	RTY
Station performance	TRY
Fixture comparison	TRY
Operator performance	TRY
Repair line efficiency	TRY
Trend analysis	FPY or TRY (per context)
Top runners	Unit count by product (per process)

Checklist for Yield Questions

1. Is the process/test_operation specified?
 2. Is unit-based (FPY) or report-based (TRY) yield appropriate?
 3. Are we dealing with a retest-only station? (Use TRY)
 4. Is volume context needed? (Top runners = high volume)
 5. Should RTY be calculated? (Overall quality across processes)
-

API Tips

Discover Processes for a Product

```
yield_tool.analyze(YieldFilter(  
    part_number="WIDGET-001",  
    perspective="by operation",  
    days=30  
))
```

Find Top Runners for a Process

```
yield_tool.analyze(YieldFilter(  
    test_operation="FCT",  
    perspective="by product",  
    days=30  
))  
# Sort results by unit_count descending
```

Get RTY Components

Query each process individually and multiply FPYs:

```
# Get all process yields  
result = yield_tool.analyze(YieldFilter(  
    part_number="WIDGET-001",  
    perspective="by operation",  
    days=30  
))  
  
# RTY = product of all FPY values  
rty = 1.0  
for process in result.data:  
    rty *= (process.first_pass_yield / 100)  
rty *= 100 # Convert back to percentage
```

Dimensional Analysis (Failure Mode Detection)

The Bridge to Root Cause

Between top-level yield analysis and detailed root cause investigation lies **dimensional analysis** - systematically comparing yields across different configurations to detect failure modes.

What is Dimensional Analysis?

Query yield data with additional dimensions (grouping factors), then compare yields to identify which configurations correlate with low yield.

Key Insight: If yield varies significantly across a dimension, that dimension is likely related to the failure mode.

Available Dimensions

Dimension	What It Reveals
stationName	Equipment issues (calibration, wear)
operator	Training/technique differences
fixtureId	Fixture wear or contamination
batchNumber	Component lot issues (supplier, incoming)
location	Environment/line differences
swFilename	Test program differences
swVersion	Test version differences
period	Drift over time

Common Failure Mode Patterns

Pattern	Typical Cause	Investigation
Station-specific	Equipment problem	Calibration, maintenance, environment
Batch-specific	Component defect	Incoming inspection, supplier quality
Operator-specific	Training gap	Standardize procedures, retrain
Fixture-specific	Fixture wear	Maintenance, contact cleaning
Time-based trend	Drift	Preventive maintenance, SPC
Software-specific	Test change	Version comparison, rollback

Analysis Workflow

1. DETECT: "FCT yield dropped from 95% to 88%"
→ Use `yield_tool` with `perspective="trend"`
2. DIAGNOSE: "What's causing it?"
→ Use `dimensional_analysis_tool`
→ Compare yield across stations, batches, operators, etc.
3. ISOLATE: "Station-3 has 75% FPY, others have 94%"
→ Found the failure mode!
4. ROOT CAUSE: "Why is Station-3 failing?"
→ Use `test_step_analysis` to find failing steps
→ Use `measurement_tool` to check Cpk

Example: Finding a Failure Mode

```
# Step 1: Notice yield problem
yield_result = yield_tool.analyze(YieldFilter(
    part_number="WIDGET-001",
    test_operation="FCT",
    days=30
))
# Result: FPY=88% (below target of 95%)

# Step 2: Dimensional analysis
failure_modes = dimensional_analysis_tool.analyze(FailureModeFilter(
    part_number="WIDGET-001",
    test_operation="FCT",
    days=30
))
# Result:
#   Station-3: FPY=75% (-13% vs baseline) CRITICAL
#   Batch-042: FPY=82% (-6% vs baseline) HIGH
#   All other dimensions: within normal range

# Step 3: Investigate Station-3 specifically
station_3_steps = test_step_analysis_tool.analyze(
    part_number="WIDGET-001",
    test_operation="FCT",
    station_name="Station-3"
)
# Result: "Voltage Test" step has 25% failure rate
```

Statistical Significance

Not all yield variations are meaningful. The tool considers:

Factor	Weight
Magnitude	How much below baseline?
Sample size	More units = higher confidence
Consistency	Does it repeat over time?

Significance Levels:

- ● CRITICAL: >10% below baseline, high confidence
- ● HIGH: 5-10% below baseline
- ● MODERATE: 2-5% below baseline
- ● LOW: <2% below baseline

Misc Info as Dimensions

WATS supports custom properties via **misc_info**. Common examples:

Misc Info	Use Case
Component lot	Track specific component batches
Firmware version	Software-under-test version
Configuration	Product variants
Supplier	Supplier traceability
Chamber ID	Environmental chamber tracking

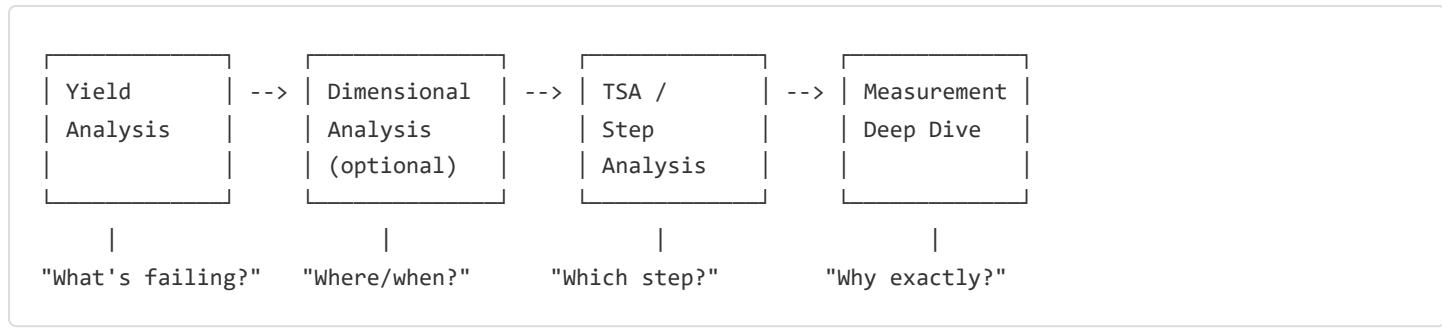
If a failure mode correlates with misc_info values, it often points directly to the root cause (e.g., a specific component lot).

Test Step Analysis (TSA)

What is TSA?

Test Step Analysis (TSA) provides step-by-step visibility into the test sequence. It shows statistics for each test step, enabling root cause identification and process capability assessment.

Position in Analysis Workflow



Key TSA Concepts

Single Product/Process Analysis

TSA is designed for ONE product in ONE process at a time.

- Different products have different test sequences
- Different processes test different things
- Mixing sequences leads to confusing merged results

Data Integrity Check

Before analyzing, TSA checks for potential issues:

Check	Warning
Multiple SW versions	Different test programs may have different sequences
Multiple revisions	Different product revisions may have different tests

Recommendation: Filter to specific `sw_filename` or `revision` for clean analysis.

Sequence Merging Behavior

When data includes multiple test sequences:

1. **Identical step paths** are merged (statistics combined)
2. **Different step paths** are kept separate
3. **Sample counts may vary** per step (not necessarily a problem)

This allows comparing related tests but can be confusing if unintended.

TSA Statistics Explained

Step Failure Statistics

Field	Meaning	Priority
step_failed_count	Step reported failure	Medium
step_error_count	Step had an error	Medium
step_terminated_count	Test terminated at this step	Medium
step_caused_uut_failed	Step CAUSED unit to fail	HIGH
step_caused_uut_error	Step caused unit error	HIGH

Critical distinction: A step can FAIL (report failure) without being the CAUSE of unit failure. The `step_caused_uut_*` fields identify root cause steps.

Process Capability (Cp/Cpk)

For measurements, TSA provides process capability indices:

Metric	Definition	Interpretation
Cpk	Process capability index	Actual capability (considers centering)
Cp	Process capability potential	Potential if perfectly centered
Cp_lower	Capability vs low limit	Lower spec margin
Cp_upper	Capability vs high limit	Upper spec margin

Cpk Thresholds

Cpk Value	Status	Action Required
≥ 1.33	CAPABLE	Process is good, monitor
1.0-1.33	MARGINAL	Improvement needed, prioritize
0.67-1.0	INCAPABLE	Action required, risk of failures
< 0.67	CRITICAL	URGENT - high defect rate expected

Industry Standard: $Cpk \geq 1.33$ ensures 3-sigma coverage on both sides.

Other Statistics

Field	Description
avg , min , max	Measurement statistics
stdev	Standard deviation (process variation)
sigma_high_3 , sigma_low_3	$\pm 3\sigma$ limits for control charts
step_time_avg , step_time_min , step_time_max	Timing statistics

Analysis Priority

When reviewing TSA results, investigate in this order:

1. ● CRITICAL: Steps Causing Unit Failures

Steps where `step_caused_uut_failed > 0` are root causes:

```
# These steps CAUSED units to fail - investigate first!
critical_steps = [s for s in steps if s.step_caused_uut_failed > 0]
critical_steps.sort(key=lambda x: x.step_caused_uut_failed, reverse=True)
```

2. ⚠ Cpk Concerns

Measurements with Cpk below threshold indicate capability issues:

```
# Measurements with poor capability
cpk_concerns = [s for s in steps if s.cpk and s.cpk < 1.33]
cpk_concerns.sort(key=lambda x: x.cpk) # Worst first
```

3. High Failure Rate Steps

Steps with high failure rates (may not be root cause):

```
# High failure rate but check if they cause unit failures
high_fail = [s for s in steps if s.pass_rate < 95 and s.step_count >= 10]
```

Common TSA Workflows

Finding Root Cause of Failures

```
# Step 1: Get step analysis
result = step_analysis_tool.analyze(StepAnalysisInput(
    part_number="PCBA-001",
    test_operation="FCT",
    days=30
))

# Step 2: Check critical steps (caused unit failures)
for step in result.critical_steps:
    print(f"{step.step_name}: {step.caused_unit_fail} unit failures")

# Step 3: If measurement, check capability
if step.cpck:
    print(f" Cpk: {step.cpck:.2f} ({step.cpck_status})")
```

Process Capability Assessment

```
# Get overall capability picture
summary = result.overall_summary

print(f"Total measurements: {summary.total_measurements}")
print(f"Average Cpk: {summary.avg_cpck:.2f}")
print(f" Capable ( $\geq 1.33$ ): {summary.capable_count}")
print(f" Marginal (1.0-1.33): {summary.marginal_count}")
print(f" Incapable ( $< 1.0$ ): {summary.incapable_count}")
```

Investigating Specific Step

After identifying a problem step:

```
# Deep dive on measurement
measurement_result = measurement_tool.analyze(MeasurementFilter(
    part_number="PCBA-001",
    test_operation="FCT",
    measurement_path="Main/Voltage Test/Output",
    days=30
))
# Get distribution, histogram, outliers...
```

TSA vs Dimensional Analysis

TSA	Dimensional Analysis
WHICH step is failing	WHERE/WHEN failures happen
Step-level statistics	Configuration comparisons
Process capability	Failure mode detection
Single product/process	Yield across dimensions

Use Together: Dimensional analysis finds the failure mode (e.g., "Station-3"), then TSA finds the specific step causing issues on that station.

Best Practices

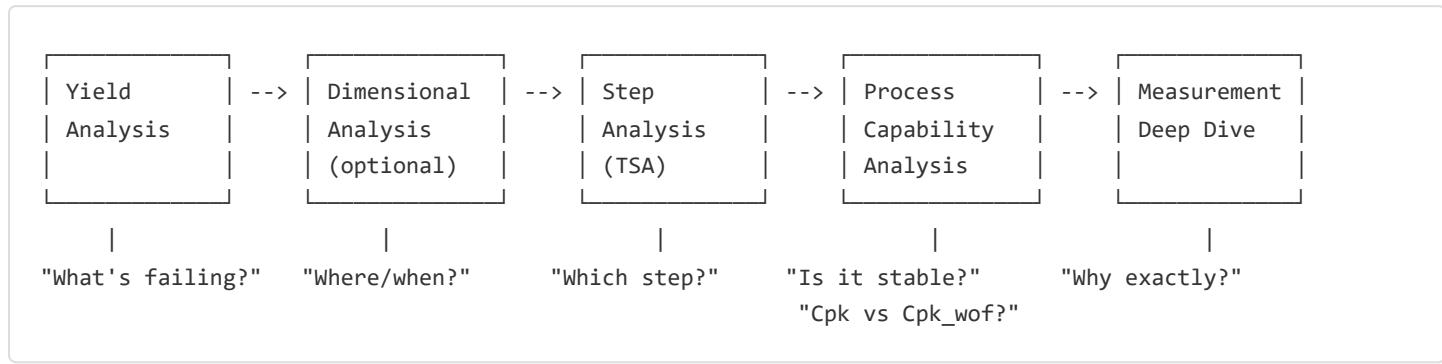
1. **Always filter to single product + process** for meaningful results
2. **Check data integrity** - alert on multiple SW versions/revisions
3. **Focus on step_caused_uut_failed** - these are the true root causes
4. **Use Cpk thresholds** - prioritize measurements below 1.33
5. **Sequence merging is OK** if understood - different step counts may be expected

Process Capability Analysis (Advanced)

Process Capability Analysis builds on TSA to provide deeper statistical assessment. Use this when TSA identifies measurements with Cpk concerns and you need to understand:

1. **Is the process stable?** (If not, Cpk is meaningless)
2. **How are failures affecting capability?**
3. **Are there hidden modes in the data?**

Position in Analysis Workflow

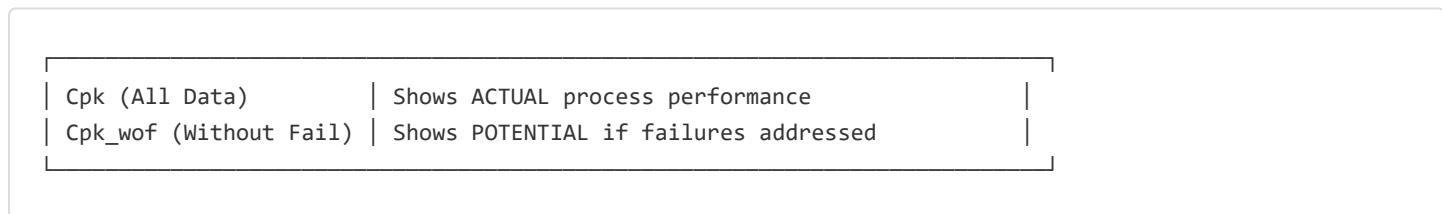


Dual Cpk Analysis (Critical!)

WATS provides TWO Cpk datasets for each measurement:

Dataset	Name	Description
Cpk	All Data	Includes ALL measurements including failures
Cpk_wof	Without Failures	Excludes failed measurements - shows "good" data

Why Two Datasets?



Interpreting the Difference

Scenario	Interpretation	Action
Cpk ≈ Cpk_wof	Process is stable, failures not distorting capability	Focus on reducing variation
Cpk << Cpk_wof	Failures significantly impact capability	Address failure root cause FIRST
Cpk >> Cpk_wof	Unusual - failures catching bad units correctly	Verify limits are correct

Example:

- Cpk = 0.9, Cpk_wof = 1.5
- Ratio = 1.67 (significant!)
- **Conclusion:** Failures are severely impacting capability. Fix the failure mode first, and capability should improve to ~1.5.

Other Capability Metrics (_wof variants)

All these statistics have "without failure" variants:

Metric	With Failures	Without Failures
Cpk	cpk	cpk_wof
Cp	cp	cp_wof
Average	avg	avg_wof
Std Dev	stdev	stdev_wof
Min/Max	min , max	min_wof , max_wof
σ Limits	sigma_high_3 , sigma_low_3	sigma_high_3_wof , sigma_low_3_wof

Stability Assessment (MUST CHECK FIRST!)

Before trusting ANY Cpk number, verify process stability.

An unstable process makes Cpk meaningless - the number will change over time.

What Makes a Process Stable?

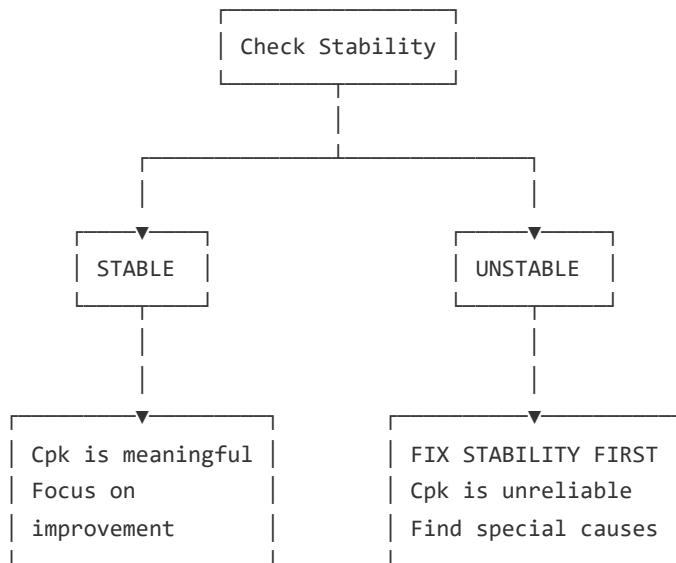
A stable process has:

- Consistent mean (no drift)
- Consistent variation (σ stays constant)
- No unusual patterns (trends, cycles, shifts)
- Random variation only (normal distribution)

Stability Red Flags

Issue	Description	Detection
Trend	Mean drifting up or down over time	Compare early vs late data
Shift	Sudden mean change	Large mean difference between periods
Outliers	Data points beyond 3σ	Values outside control limits
High Variance	6σ spread > spec range	Process spread exceeds tolerance
Bimodal	Two populations mixed	σ_{wof} much smaller than σ

Stability Decision Flow



Hidden Mode Detection

Beyond basic capability, look for these hidden issues:

1. Centering Issues ($C_p >> C_{pk}$)

If C_p is much higher than C_{pk} , the process could be capable but is off-center:

Example:
 $C_p = 1.8$ (good potential)
 $C_{pk} = 0.9$ (poor actual)
Ratio = 2.0

→ Process is off-center from spec midpoint
→ Centering adjustment could double C_{pk} !

2. Approaching Specification Limits

Check sigma margin to specification limits:

Sigma Margin	Risk Level	Action
$> 3\sigma$	Low	Process is well-centered
$2-3\sigma$	Medium	Monitor closely
$< 2\sigma$	High	Failures likely, adjust process

3. Bimodal Distribution

Signs of two populations mixed:

- σ_{wof} is much smaller than σ
- Mean_wof differs significantly from Mean
- Distribution may have two peaks

Root Cause: Often a machine, operator, or component lot causing different behavior.

4. High Variance Relative to Specification

Check if process spread fits within specification:

Process Spread = 6σ

Spec Range = Upper Limit - Lower Limit

If $6\sigma > \text{Spec Range}$ → Process inherently incapable

Improvement Priority Matrix

Use this to prioritize improvement efforts:

Priority	Criteria	Action
● CRITICAL	Cpk < 0.67 OR Process unstable	Immediate action
● HIGH	Cpk < 1.0 OR Approaching limits	Address soon
● MEDIUM	1.0 ≤ Cpk < 1.33 OR Centering issue	Plan improvement
● LOW	Cpk ≥ 1.33 AND Stable	Monitor only

Dimensional Considerations

Important: For dimensional analysis within process capability:

- TSA returns aggregate statistics across ALL data
- To analyze by dimension (station, operator, SW version), you must:
 - Filter the data to the specific dimension value
 - Re-run the analysis
 - Compare results across dimensions

Example workflow:

```

# Compare capability across stations
for station in ["Station-1", "Station-2", "Station-3"]:
    result = capability_tool.analyze(ProcessCapabilityInput(
        part_number="PCBA-001",
        test_operation="FCT",
        station_name=station # Filter to specific station
    ))
    print(f"{station}: Cpk={result.avg_cpk_all:.2f}, "
          f"Stable={result.stable_count}/{result.measurements_analyzed}")

```

Process Capability vs TSA

Feature	TSA	Process Capability
Scope	All steps overview	Detailed per measurement
Cpk Analysis	Basic (single value)	Dual (with/without failures)
Stability	Not checked	Full stability assessment
Hidden Modes	Not detected	Detects trends, outliers, etc.
When to Use	First pass analysis	Deep dive on concerns

Best Practices

- 1. Check stability BEFORE trusting Cpk** - unstable processes have meaningless Cpk
- 2. Compare Cpk vs Cpk_wof** - significant difference means address failures first
- 3. Look for centering issues** - Cp >> Cpk means easy improvement available
- 4. Check sigma margins** - less than 2σ to limits is high risk
- 5. For dimensions, make separate calls** - aggregate data hides variation
- 6. Prioritize by improvement priority** - critical first, then high, etc.

Summary

The most important things to remember:

- 1. Yield is per process** - Always clarify which test operation
- 2. RTY for overall quality** - Multiply FPYs across all processes
- 3. TRY for equipment/operator** - Especially for retest stations
- 4. Top runners are per process** - Volume varies by test operation

5. **Unit Inclusion Rule** - First run determines inclusion
 6. **Dimensional analysis for failure modes** - Compare yields across configurations
 7. **TSA for root cause** - Find which STEP is causing failures
 8. **Process Capability for deep dive** - Stability, dual Cpk, hidden modes
 9. **Cpk vs Cpk_wof** - Compare to understand failure impact
 10. **Check stability first** - Unstable process makes Cpk meaningless
 11. **Cpk ≥ 1.33 is capable** - <1.0 needs action, <0.67 is critical
 12. **step_caused_uut_failed is key** - Identifies true root cause steps
 13. **Follow the workflow** - Yield → Dimensions → Steps → Capability → Measurements
-

Source: [docs/llm-converter-guide.md](#)

LLM Quick Reference - ReportBuilder for pyWATS

For LLMs Implementing Converters

This is your cheat sheet for creating pyWATS converters. **You only need to customize the parsing logic** - everything else is standard.

Standard Converter Template

```
from pywats.tools import ReportBuilder
from pywats_client.converters.file_converter import FileConverter
from pywats_client.converters.models import (
    ConverterSource, ConverterResult, ValidationResult
)

class YourConverter(FileConverter):
    """Your converter description"""

    @property
    def name(self) -> str:
        return "Your Converter Name"

    @property
    def version(self) -> str:
        return "1.0.0"

    @property
    def file_patterns(self) -> List[str]:
        return ["*.yourext"]

    def convert(self, source: ConverterSource, context) -> ConverterResult:
        try:
            # STEP 1: Parse file (ONLY CUSTOM PART)
            data = self._parse_file(source.read_text())

            # STEP 2: Create builder (STANDARD)
            builder = ReportBuilder(
                part_number=data["part_number"],
                serial_number=data["serial_number"]
            )

            # STEP 3: Add steps (STANDARD)
            for test in data["tests"]:
                builder.add_step(
                    name=test["name"],
                    value=test["value"],
                    unit=test.get("unit"),
                    low_limit=test.get("low_limit"),
                    high_limit=test.get("high_limit"),
                    group=test.get("group")
                )

            # STEP 4: Build (STANDARD)
            return ConverterResult.success_result(
                report=builder.build()
            )
        except Exception as e:
            return ConverterResult.error_result(str(e), e)

    def _parse_file(self, content: str) -> dict:
        """
```

CUSTOMIZE THIS METHOD FOR YOUR FORMAT

```
Must return:  
{  
    "part_number": "PN-001",  
    "serial_number": "SN-001",  
    "tests": [  
        {  
            "name": "Test Name",  
            "value": 5.0,          # Can be: float, int, bool, str, list  
            "unit": "V",           # Optional  
            "low_limit": 4.5,      # Optional  
            "high_limit": 5.5,     # Optional  
            "group": "Power Tests" # Optional (creates sequences)  
        },  
        ...  
    ]  
}  
"""  
# YOUR PARSING LOGIC HERE  
pass
```

Quick Add Step Reference

Basic Patterns

```
# Numeric test with limits  
builder.add_step("Voltage", 5.0, unit="V", low_limit=4.5, high_limit=5.5)  
  
# Numeric test without limits (LOG only)  
builder.add_step("Temperature", 25.3, unit="C")  
  
# Boolean test  
builder.add_step("Power OK", True)  
  
# String test  
builder.add_step("Serial Number", "ABC123")  
  
# Multi-value numeric  
builder.add_step("Calibration", [1.2, 1.3, 1.1], unit="mV")  
  
# With grouping (creates sequence)  
builder.add_step("VCC", 3.3, unit="V", group="Power Tests")
```

Type Inference Rules

Value Type	Result
True or False	Boolean step
5.0 or 100	Numeric step
"ABC"	String step
[1.0, 2.0, 3.0]	Multi-numeric step
[True, False]	Multi-boolean step
["A", "B"]	Multi-string step

Status Calculation

```
# In range → Pass (auto)
builder.add_step("Test", 5.0, low_limit=4.0, high_limit=6.0)

# Out of range → Fail (auto)
builder.add_step("Test", 10.0, low_limit=4.0, high_limit=6.0)

# Manual override
builder.add_step("Test", 5.0, status="F") # Force fail
```

Parsing Examples

CSV Format

```
def _parse_file(self, content: str) -> dict:
    import csv
    lines = content.strip().split('\n')
    reader = csv.DictReader(lines)

    tests = []
    header = {}

    for i, row in enumerate(reader):
        if i == 0: # First row has header info
            header = {
                "part_number": row.get("pn", "PN-UNKNOWN"),
                "serial_number": row.get("sn", "SN-UNKNOWN")
            }

        tests.append({
            "name": row["test_name"],
            "value": float(row["value"]) if row["value"].replace('.','').isdigit() else row["value"],
            "unit": row.get("unit"),
            "low_limit": float(row["low_limit"]) if row.get("low_limit") else None,
            "high_limit": float(row["high_limit"]) if row.get("high_limit") else None
        })

    return {"part_number": header["part_number"], "serial_number": header["serial_number"], "tests": tests}
```

JSON Format

```
def _parse_file(self, content: str) -> dict:
    import json
    data = json.loads(content)

    tests = []
    for test_name, test_data in data["tests"].items():
        tests.append({
            "name": test_name,
            "value": test_data["value"],
            "unit": test_data.get("unit"),
            "low_limit": test_data.get("low_limit"),
            "high_limit": test_data.get("high_limit")
        })

    return {
        "part_number": data["pn"],
        "serial_number": data["sn"],
        "tests": tests
    }
```

XML Format

```
def _parse_file(self, content: str) -> dict:
    import xml.etree.ElementTree as ET
    root = ET.fromstring(content)

    tests = []
    for test in root.findall("./Test"):
        tests.append({
            "name": test.get("name"),
            "value": float(test.find("Value").text),
            "unit": test.get("unit"),
            "low_limit": float(test.find("LowLimit").text) if test.find("LowLimit") is not None else
None,
            "high_limit": float(test.find("HighLimit").text) if test.find("HighLimit") is not None else
None
        })

    return {
        "part_number": root.find("./PartNumber").text,
        "serial_number": root.find("./SerialNumber").text,
        "tests": tests
    }
```

Tab-Separated Text

```
def _parse_file(self, content: str) -> dict:
    lines = content.split('\n')

    header = {}
    tests = []

    for line in lines:
        line = line.strip()
        if not line or line.startswith('#'):
            continue

        # Header lines (KEY: VALUE)
        if ':' in line and '\t' not in line:
            key, value = line.split(':', 1)
            if key.strip().upper() == 'SERIAL':
                header['serial_number'] = value.strip()
            elif key.strip().upper() == 'PART':
                header['part_number'] = value.strip()

        # Test lines (tab-separated)
        elif '\t' in line:
            parts = line.split('\t')
            tests.append({
                "name": parts[0],
                "value": float(parts[1]) if parts[1].replace('.', '').replace('-', '').isdigit() else
parts[1],
                "unit": parts[2] if len(parts) > 2 and parts[2] != '-' else None,
                "low_limit": float(parts[3]) if len(parts) > 3 and parts[3] != '-' else None,
                "high_limit": float(parts[4]) if len(parts) > 4 and parts[4] != '-' else None
            })

    return {"part_number": header.get("part_number", "UNKNOWN"), "serial_number": header.get("serial_number", "UNKNOWN"), "tests": tests}
```

Common Mistakes to Avoid

✗ DON'T: Build reports manually

```
report = UUTReport(...)
root = report.get_root_sequence_call()
root.add_numeric_step(...) # Too many parameters, easy to mess up
```

DO: Use ReportBuilder

```
builder = ReportBuilder(pn, sn)
builder.add_step(name, value, unit, low_limit, high_limit)
report = builder.build()
```

DON'T: Calculate status manually

```
if value >= low_limit and value <= high_limit:
    status = "P"
else:
    status = "F"
builder.add_step(name, value, status=status)
```

DO: Let the builder calculate it

```
builder.add_step(name, value, low_limit=low_limit, high_limit=high_limit)
# Status calculated automatically
```

DON'T: Manually create sequences

```
root = report.get_root_sequence_call()
power_seq = root.add_sequence_call("Power Tests")
power_seq.add_numeric_step(...)
```

DO: Use group parameter

```
builder.add_step("VCC", 3.3, group="Power Tests")
builder.add_step("VDD", 1.8, group="Power Tests")
# Sequences created automatically
```

Validation Checklist

Before submitting your converter code, verify:

- [] Uses `from pywats.tools import ReportBuilder`
- [] Only `_parse_file()` method contains custom logic
- [] Returns dict with `part_number`, `serial_number`, `tests`
- [] Each test has `name` and `value` (minimum)

- [] Uses `builder.add_step()` not manual `UUTReport` construction
 - [] Lets builder infer types, operators, status
 - [] Returns `ConverterResult.success_result(report=builder.build())`
 - [] Has try/except with `ConverterResult.error_result()` on exception
-

Complete Minimal Example

```
from pywats.tools import ReportBuilder
from pywats_client.converters.file_converter import FileConverter
from pywats_client.converters.models import ConverterSource, ConverterResult

class MinimalConverter(FileConverter):
    @property
    def name(self) -> str:
        return "Minimal Converter"

    @property
    def version(self) -> str:
        return "1.0.0"

    @property
    def file_patterns(self) -> List[str]:
        return ["*.txt"]

    def convert(self, source: ConverterSource, context) -> ConverterResult:
        try:
            # Parse
            lines = source.read_text().split('\n')
            pn = lines[0].split(':')[1].strip()
            sn = lines[1].split(':')[1].strip()

            # Build
            builder = ReportBuilder(pn, sn)
            for line in lines[2:]:
                if '\t' in line:
                    name, value = line.split('\t')
                    builder.add_step(name, float(value))

            # Return
            return ConverterResult.success_result(report=builder.build())
        except Exception as e:
            return ConverterResult.error_result(str(e), e)
```

That's it! **Only 20 lines of actual code.**

Need Help?

- **Documentation:** See `docs/usage/REPORT_BUILDER.md`
 - **Examples:** See `examples/report/report_builder_examples.py`
 - **Template:** See `converters/simple_builder_converter.py`
-

pyWATS Documentation © Virinco AS

Generated on 2026-01-26