CalculationClient DLL API Reference

Native C and C++ integration for TFCompanion Calculation Server | Windows x64 | 9 November 2026

This reference describes all 19 exported functions in CalculationClient.dll. It is intended for developers submitting reflectance spectra for calculation and receiving results without a connected spectrometer. The public interface is declared in include/CalculationClient.h.

Required setup before calculation

Set either a filmstack with `TFC_SetFilmstack` or a measurement recipe with `TFC_SetMeasurementRecipe` before calling `TFC_Calculate`. A measurement recipe includes a filmstack, so calling both setters is not required. Check that the selected setter returns `TFC_OK`.

A recipe also supplies calculation conditions. Selecting a filmstack alone changes the model and leaves the current server calculation conditions in place. Use `TFC_SetCalculationConditions` when particular conditions are required. Conditions alone do not replace the filmstack or recipe requirement.

Server compatibility: filmstack-only calculation requires the updated TFCompanion handler. With no recipe loaded it uses normal-incidence reflectance at 0 degrees. Older builds reject this path with exception 8; update tfcomp.jar. See Calculation setup functions.

Connection and calculation sequence

1. Call TFC_Create to start the local TCP and UDP result listeners.

2. Call TFC_Connect to connect to TFCompanion and request its DataClient callback connection to those listeners.

3. Set either the measurement recipe or filmstack. Apply any required conditions, parameter changes, or layer changes.

4. Call TFC_Calculate with wavelength/reflectance pairs. Keep the returned calculation ID.

5. Drain TFC_PollEvent regularly and match result IDs to submissions. Check result exception, warning, and quality flags.

6. Stop other callers, then call TFC_Destroy to close connections and stop the receiver.

Calling conventions and error codes

Build and deployment

Build the caller for x64. Include `CalculationClient.h`, link `lib/x64/CalculationClient.lib`, and deploy `CalculationClient.dll` beside the executable. Use the header, import library, and DLL from the same SDK release. Do not define TFC_BUILD in client code.

The interface has C linkage and uses `TFC_CALL` (`__cdecl`). Preserve the header field types, order, and default MSVC structure alignment; do not apply custom packing. The client DLL requires no Java, JNI, .NET, or Qt installation. It uses Windows networking and a statically linked C++ runtime.

Common argument and ownership rules

Unless explicitly noted otherwise, client must be a live handle returned by TFC_Create. Passing a stale or fabricated non-null pointer is invalid. Strings are NUL-terminated UTF-8. Endpoints are numeric IPv4 addresses, not DNS names or IPv6 addresses. Input pointers must remain valid until the call returns.

For functions returning a TFC_Error code, reply may be NULL. When supplied, the DLL initializes the entire TFC_Reply and sets reply.code equal to the return value. Use a separate reply/event buffer for each concurrent caller. No allocated buffer crosses the API: the caller owns all input and output structures; only TFC_Destroy frees a client handle.

Value	Name	Meaning
0	`TFC_OK`	The local operation or server command succeeded. Calculation results still arrive separately.
1	`TFC_INVALID_ARGUMENT`	A required argument, range, or local call state is invalid.
2	`TFC_NETWORK_ERROR`	Socket, bind, connection, send, receive, or peer-close failure.
3	`TFC_TIMEOUT`	The configured network deadline expired; request outcome may be unknown.
4	`TFC_PROTOCOL_ERROR`	Malformed command reply or a mismatched response function.
5	`TFC_SERVER_ERROR`	The server reported an exception, failed confirmation, or protocol error.
6	`TFC_NOT_CONNECTED`	The command connection or local receiver is not running.
7	`TFC_INTERNAL_ERROR`	A native exception or another internal failure was caught.

Concurrency and recovery

Calls that use the command connection, including Connect and Disconnect, are serialized per handle. A waiting caller can spend time waiting for that lock in addition to its own network deadline. PollEvent can run concurrently with commands; it briefly locks the local event queue and does not wait for network data.

Network failures, timeouts, and malformed command replies close the command connection. A server rejection normally leaves it open; a failed Connect closes it. The DLL never automatically retries a request. Reconnect and reconcile pending results before resubmitting a calculation or state-changing command.

Configuration and command reply structures

TFC_Handle and TFC_Config

TFC_Handle is an opaque pointer. Initialize it to NULL and obtain it through TFC_Create. The configuration is read during that call and is not retained by pointer.

TFC_Config member	Type and meaning
`bind_ipv4`	const char*: local listener address. NULL, empty, or 0.0.0.0 binds all IPv4 interfaces.
`data_port`	uint16_t: shared TCP and UDP listener port. 0 requests an available port; query the selected value with TFC_DataPort.
`timeout_ms`	uint32_t: 100..60000 milliseconds. Network timeout for command operations and incomplete data frames. No implicit default when the caller supplies zero.

TFC_Reply

Fields are zero-initialized for each call. Read operation-specific fields only after TFC_OK. A failed response can still contain useful message and server status fields, but they may be absent if the reply was incomplete.

Member	Type and meaning
`code`	int32_t: DLL error code; same value as the function return.
`system_status`	uint16_t: raw server system status from a normal command reply.
`warning`	uint16_t: raw server warning. A warning alone does not cause a nonzero return.
`exception`	uint16_t: server exception or protocol-error code. Nonzero server exceptions cause TFC_SERVER_ERROR.
`server_version[11]`	char: ten server-version bytes followed by a NUL terminator.

Member	Type and meaning
`message[256]`	char: NUL-terminated diagnostic text. Read it on failure and after Connect.
`data_id`	uint64_t: accepted calculation ID from TFC_Calculate; match it to TFC_Event.data_id.
`value`	float: parameter value from TFC_GetParameter, or average calculation time in milliseconds from TFC_GetExecutionTime.
`free_mb`, `used_mb`	float: free and used server JVM heap from TFC_GetMemoryStatus. The server divides bytes by 1048576 (MiB).
`queue_size`	uint32_t: waiting calculation count from TFC_GetQueueSize; the underlying reply carries a 16-bit count.

Server system, warning, and exception codes belong to TFCompanion and are separate from TFC_Error. Preserve their numeric values for diagnostics; this DLL does not define their full enumeration.

Event and result structures

TFC_PollEvent returns one TFC_Event at a time. Fields are meaningful according to event.type and, for results, event.function. TFC_Parameter contains `char name[256]` and `float value`; names are NUL-terminated and values use the server-configured units.

Value and event type	Meaning
1 `TFC_RESULT`	A calculation result or calculation error, identified by data_id.
2 `TFC_HELLO`	Greeting received from the remote DataClient; message contains its text.
3 `TFC_DATA_CONNECTED`	An accepted TCP callback connection. UDP has no corresponding connection event.
4 `TFC_DATA_DISCONNECTED`	TCP callback closed; the local listener remains active.
5 `TFC_DATA_ERROR`	A malformed result, incomplete frame timeout, or receiver failure; inspect message.

TFC_Event member	Type and meaning
type, function	int32_t event type; uint32_t data protocol function. Functions 86, 87, and 88 are results; 83 is a greeting.
data_id	uint64_t: ID of the submitted spectrum, for result events.
exception, warning	uint16_t: server result status when supplied by that result format.
high_mse, outlier	uint8_t: raw server filter flags supplied by function 88; preserve their values.
has_mse, mse	uint8_t presence flag and float fit error. Read mse only when has_mse is nonzero.
reserved	uint8_t: reserved, currently zero.
parameter_count	uint32_t: number of valid TFC_Parameter entries; 0..TFC_MAX_PARAMETERS (36).
parameters[36]	TFC_Parameter array. Read only entries below parameter_count.
raw_size, raw[255]	uint32_t length and uint8_t array: exact received payload, excluding framing, for greeting/result diagnostics.
dropped_events	uint64_t: overflow losses since the previous successful event dequeue; resets when reported.
message[256]	char: NUL-terminated greeting or diagnostic; normally empty for numerical results.

Function 86 carries named parameters without MSE, or a status-only error with no parameters. Function 87 adds MSE. Function 88 adds status, quality flags, and MSE. Zero in a field omitted by the selected format is not proof that the server explicitly reported zero.

Client lifetime functions

TFC_Create

```
int TFC_CALL TFC_Create(
    const TFC_Config* config,
    TFC_Handle* client,
    TFC_Reply* reply);
```

Creates a client and starts both local TCP and UDP data listeners plus the receiving thread. No command connection is made yet. Call this before TFC_Connect.

config and client are required pointers. Configure the listener address, port, and timeout under Configuration and command reply structures. reply is optional. Returns TFC_OK and writes a live handle to *client; on failure, *client is NULL. Both protocols must bind successfully to the same port. An occupied or unavailable endpoint produces a network error; all partially created resources are released.

TFC_Destroy

```
void TFC_CALL TFC_Destroy(
    TFC_Handle client);
```

Stops the receiving thread, waits for it to exit, closes the command and listener sockets, and releases the handle. NULL is a no-op. There is no return value. Stop and join all other callers first, including any polling thread. The handle must not be used after this call. This does not send a remote server shutdown or remote DataClient stop command.

TFC_DataPort

```
uint16_t TFC_CALL TFC_DataPort(
    TFC_Handle client);
```

Returns the actual local TCP/UDP listener port, including the selected port when config.data_port was zero. NULL returns 0. This is a local accessor and sends no server command. The selected port remains fixed for the lifetime of the handle; recreate the client to change its binding.

TFC_IsConnected

```
int TFC_CALL TFC_IsConnected(
    TFC_Handle client);
```

Returns 1 when the DLL considers the command connection active and the receiver thread is running; otherwise returns 0. NULL returns 0. This is a local state check, not a network probe or proof that result traffic is arriving. A remote command-socket closure may remain undetected until the next command.

Lifetime example

```
TFC_Config config{nullptr, 5001, 5000};
TFC_Handle client = nullptr;
TFC_Reply reply{};
int rc = TFC_Create(&config, &client, &reply);
if (rc == TFC_OK) {
    // Connect, configure, calculate, and poll results.
    // Stop all concurrent callers before destroying the handle.
    TFC_Destroy(client);
    client = nullptr;
}
```

A client listening on all interfaces accepts result traffic only from the configured calculation server IPv4 address. The receiver remains independent of the command connection, including after a command disconnect.

Connection and event functions

TFC_Connect

```
int TFC_CALL TFC_Connect(
    TFC_Handle client,
    const char* server_ipv4,
    uint16_t port,
    const char* callback_ipv4,
    TFC_Reply* reply);
```

client must be live, disconnected, and have a running receiver. server_ipv4 is a required numeric unicast IPv4 address; port is 1..65535. The usual TFCompanion calculation port is 500. This is the dedicated calculation endpoint, not standard Modbus TCP port 502.

callback_ipv4 is the address of this client PC that the server can reach. NULL or an empty string uses the command socket's local IPv4 address. The callback port is TFC_DataPort(client). The function opens the command connection and sends START_DATA_CLIENT automatically. TFCompanion's PLC properties select TCP or UDP for its DataClient; the DLL listens on both.

Returns TFC_OK only after DataClient startup is acknowledged. It does not wait for a greeting or prove callback delivery. Check PollEvent for a greeting/result and allow inbound traffic to the callback port. TCP connection establishment and the startup command each have their own configured network timeout. Disconnect before calling Connect again on an active handle.

TFC_Disconnect

```
void TFC_CALL TFC_Disconnect(
    TFC_Handle client);
```

Closes the command connection and marks it disconnected. It waits for the per-handle command lock if a command is running. NULL is a no-op; there is no return value. Local listeners and queued events remain available, allowing pending results from the configured server to arrive and be polled.

TFC_PollEvent

```
int TFC_CALL TFC_PollEvent(
    TFC_Handle client,
    TFC_Event* event);
```

client and event are required. Returns 1 after copying and removing the oldest event, 0 when the queue is empty, and -1 for a NULL handle or event pointer. The output is unchanged for 0 and -1. This return convention is not TFC_Error.

The call does not wait for network traffic and can run alongside commands. Use one logical consumer to avoid splitting related results across consumers. The queue holds 256 events; when full, the oldest event is discarded. A successful poll reports accumulated losses in event.dropped_events. Poll regularly and process all available events.

Reconnect behavior

TFCompanion can retain its DataClient across command disconnects. A new startup request may acknowledge the existing DataClient without changing its callback destination or sending another greeting. Keep the same local receiver alive for command reconnects. If the application restarts or its

callback address/port changes and results stop, restart the server DataClient or Calculation Server as appropriate.

If the receiver thread stops, recreate the client; reopening only the command connection cannot restart it. Queued events can survive command reconnects, so continue matching result IDs. UDP delivery may lose or reorder packets; TCP is preferable when reliable result delivery is required.

Calculation setup functions

Before calculation, successfully set either a filmstack or a measurement recipe. A measurement recipe includes the filmstack; it is not necessary to call both setters. The DLL forwards commands and does not maintain a local model-ready flag, so the calling application must enforce this sequence.

Server compatibility: install the TFCompanion build containing the filmstack-only handler fix in tfcomp.jar. Older builds require a measurement-recipe parser and return exception 8 (REQUEST_CONTENT_EXCEPTION) without one. The updated handler uses normal-incidence reflectance at 0 degrees when no recipe is loaded; an existing recipe retains its spectrum definition.

Verified on 9 November 2026: filmstack-only calculation with oxide2000 and recipe-based calculation with MTest10 both completed successfully after separate server restarts. Each test submitted 1,772 reflectance points and received a result with exception 0 and warning 0. The filmstack-only test did not select a measurement recipe; the recipe test did not set a separate filmstack.

TFC_SetMeasurementRecipe

```
int TFC_CALL TFC_SetMeasurementRecipe(
    TFC_Handle client,
    const char* name,
    TFC_Reply* reply);
```

name is the nonempty name of an existing measurement recipe in the server database, not a file path. It is limited to 65496 UTF-8 bytes and cannot contain *. A valid recipe supplies the filmstack and its calculation conditions. Returns a TFC_Error code and optional reply diagnostics. The command connection must be active.

The server can reuse a recipe already selected by the same name; do not rely on repeating the same name to reload edited database contents or discard later filmstack overrides. If precise state matters, set the intended filmstack and conditions explicitly after selecting the recipe.

TFC_SetFilmstack

```
int TFC_CALL TFC_SetFilmstack(
    TFC_Handle client,
    const char* name,
    TFC_Reply* reply);
```

name is the nonempty name of an existing filmstack in the server database, not a local file path. The same 65496-byte and no-* name rules apply. Returns a TFC_Error code and optional reply diagnostics. The command connection must be active.

Selects the model used for subsequent calculation and parameter/layer operations. On the updated server, no additional recipe is required. It does not load or clear a recipe's spectrum definition or

calculation conditions. Without a loaded recipe, geometry is normal incidence; retain suitable calculation conditions or set them explicitly.

TFC_SetCalculationConditions

```
int TFC_CALL TFC_SetCalculationConditions(
    TFC_Handle client,
    const char* xml,
    TFC_Reply* reply);
```

xml is a nonempty, NUL-terminated UTF-8 XML string, limited to 65496 bytes. The DLL forwards its contents; the server validates and applies them. Returns a TFC_Error code and optional reply diagnostics. A connected handle is required. This call alone does not select a filmstack.

```
<Calculation_Conditions>
  <algorithm>Marquardt-Levenberg</algorithm>
</Calculation_Conditions>
```

Use calculation XML appropriate to the installed server. The FFT modes Thick Film FFT and Combined FFT-ML also use fft_level, FFT_interface, CALC_ACCURACY, Calc_Preprocess_remove_trend, and Calc_total_thickness. Prefer the settings from a validated server recipe rather than incomplete XML.

Names and parameter identifiers must match the server database. Some legacy server paths decode text using Java's default charset; use ASCII names/XML or a matching UTF-8 server configuration for non-ASCII content.

Parameter and layer functions

These functions require an active command connection and a selected filmstack, directly or through a measurement recipe. They modify or query the current server model. Values and thickness use the server model's units; the DLL performs no unit conversion. Each returns a TFC_Error code with optional reply diagnostics.

TFC_SetParameter

```
int TFC_CALL TFC_SetParameter(
    TFC_Handle client,
    const char* name,
    float value,
    int calculate,
    TFC_Reply* reply);
```

name identifies an existing parameter. It must be nonempty, at most 65490 UTF-8 bytes, and contain neither * nor :. value must be finite. calculate must be 0 (fixed) or 1 (included as a fitted parameter); it does not submit a spectrum or immediately execute a calculation. Parameter bounds and model-specific validity are determined by the server.

TFC_GetParameter

```
int TFC_CALL TFC_GetParameter(
    TFC_Handle client,
    const char* name,
    TFC_Reply* reply);
```

name is a nonempty parameter identifier, at most 65496 UTF-8 bytes, without *. On TFC_OK, reply.value contains its current value; pass a reply pointer to retrieve it. The current server treats a negative

retrieved value as a failed request, so this operation cannot reliably return a legitimate negative parameter value on that server build.

TFC_AddLayer

```
int TFC_CALL TFC_AddLayer(
    TFC_Handle client,
    const char* material,
    float thickness,
    uint8_t index,
    TFC_Reply* reply);
```

material is the name of an existing server material: nonempty, at most 65490 UTF-8 bytes, with neither * nor :. thickness is finite and nonnegative. index is a uint8_t value (0..255). Index 1 inserts at the first layer position above the substrate; subsequent positive positions are one-based. Index 0, or a position beyond the current layer count, appends a layer.

TFC_RemoveLayer

```
int TFC_CALL TFC_RemoveLayer(
    TFC_Handle client,
    uint8_t index,
    TFC_Reply* reply);
```

index must be 1..127 and identify an existing one-based layer position. Index 1 removes the first layer above the substrate. Zero is invalid, and the maximum is 127 because the server reads this request index as a signed byte. Invalid model positions are rejected by the server.

Ordering state changes

Apply model changes before submitting the corresponding spectrum. The server holds shared model and calculation state. Use one controlling client per server instance and avoid changing recipes, filmstacks, parameters, layers, or conditions while calculations that depend on them are in flight.

The setter return value confirms command acceptance. It is not a fitted result; numerical fit output is received later as TFC_RESULT events after TFC_Calculate.

Submitting reflectance for calculation

TFC_Calculate

```
int TFC_CALL TFC_Calculate(
    TFC_Handle client,
    const float* pairs,
    uint32_t point_count,
    TFC_Reply* reply);
```

Prerequisite: successfully call TFC_SetFilmstack or TFC_SetMeasurementRecipe with suitable calculation conditions. A recipe includes a filmstack. Filmstack-only operation requires a compatible server; see Calculation setup functions. The command connection and local receiver must be running.

pairs points to an interleaved float array containing 2 * point_count elements: wavelength_nm, reflectance_fraction, wavelength_nm, reflectance_fraction, and so on. point_count is the number of pairs, not the number of floats. The array need only remain valid until this call returns.

Input	Required value
`pairs`	Non-NULL; contains at least 2 * point_count readable floats.
`point_count`	2..8187 inclusive (TFC_MAX_POINTS). Larger datasets are rejected, not automatically split.
Wavelength	Finite, greater than zero, strictly increasing, in nanometers.
Reflectance	Finite fraction in [0,1]. Divide percent values by 100 before calling the DLL.
Geometry	No angle column is sent by this API. Configure the intended geometry through the server model/setup.

```
const float pairs[] = {
    450.0f, 0.2411f,
    450.3f, 0.2428f,
    450.6f, 0.2400f
};
TFC_Reply reply{};
int rc = TFC_Calculate(client, pairs, 3, &reply);
```

These three points illustrate the array layout only; use a real spectrum and a matching server setup for a meaningful fit. The DLL accepts arrays, not CSV paths. CSV parsing and percent conversion in the test GUI belong to that application.

Acceptance and completed results

TFC_OK means the server accepted the submission and reply.data_id contains its ID. It does not mean the fit finished or succeeded. Continue polling and match TFC_Event.data_id. Result traffic can arrive before this function returns; the DLL queues it. Check exception, warning, parameter_count, has_mse, and available quality flags.

The configured timeout bounds command communication, not calculation completion. Choose an application-level deadline suitable for the calculation. A missing result is not a reason to automatically resubmit: the request may already be running. The inspected server generates IDs from millisecond timestamps, so avoid relying on uniqueness for multiple submissions within one millisecond.

Each result payload is limited to 255 bytes by the server protocol. TFC_MAX_PARAMETERS is a storage limit, not a guarantee that 36 named parameters fit in a packet. Long names or many returned parameters can exceed the protocol limit; configure result output accordingly.

Server query functions

These functions require a connected client but do not require setting a filmstack or recipe. They are synchronous command calls and can run concurrently with event polling. Each returns TFC_OK or a TFC_Error code. Supply reply to retrieve the returned fields.

TFC_GetStatus

```
int TFC_CALL TFC_GetStatus(
    TFC_Handle client,
    TFC_Reply* reply);
```

Reads the server version, system status, warning, and exception fields into reply.server_version, reply.system_status, reply.warning, and reply.exception. A nonzero server exception produces TFC_SERVER_ERROR. This query reports server status; it does not confirm that any specific calculation completed or that callback traffic is arriving.

TFC_GetMemoryStatus

```
int TFC_CALL TFC_GetMemoryStatus(
    TFC_Handle client,
    TFC_Reply* reply);
```

Returns reply.free_mb and reply.used_mb as floats. These describe free and used memory within the server JVM's allocated heap, calculated using 1048576 bytes per unit (MiB). They are not total physical RAM statistics for the server computer.

TFC_GetQueueSize

```
int TFC_CALL TFC_GetQueueSize(
    TFC_Handle client,
    TFC_Reply* reply);
```

Returns reply.queue_size, the count of calculations waiting in the server executor queue. It does not include work already executing. Therefore queue_size == 0 does not mean every accepted result has arrived. Use result IDs to track completion.

TFC_GetExecutionTime

```
int TFC_CALL TFC_GetExecutionTime(
    TFC_Handle client,
    TFC_Reply* reply);
```

Returns reply.value, the server's average calculation-task execution time in milliseconds. It is not the elapsed time of the most recent request, a timeout setting, or an estimate of the total waiting time for one queued submission.

Query example

```
TFC_Reply reply{};
if (TFC_GetQueueSize(client, &reply) == TFC_OK) {
    printf("Waiting calculations: %u\n",
           static_cast<unsigned>(reply.queue_size));
} else {
    fprintf(stderr, "Queue query failed: %s\n", reply.message);
}
```

Scope of this interface

The exports cover calculation setup, model editing, reflectance submission, receiving calculated results, and server diagnostics. They do not start spectrometer acquisition, perform calibration, or shut down the server. A client application does not need a connected spectrometer for these imported-data calculations.

C++ integration example

Use either setup path with a real spectrum and an existing server database name. Keep the handle and listener alive across calculations and reconnects. Run blocking calls off the GUI thread.

```
#include "CalculationClient.h"
#include <chrono>
#include <cstdio>
#include <memory>
#include <stdexcept>
#include <thread>

void calculate_once(const char* server, const char* setupName,
                    bool useRecipe, const float* pairs, uint32_t count) {
    TFC_Handle client = nullptr;
    TFC_Reply reply{};
    auto check = [&](int rc) {
        if (rc != TFC_OK) throw std::runtime_error(reply.message);
    };
    TFC_Config config{nullptr, 5001, 5000};
    check(TFC_Create(&config, &client, &reply));
    std::unique_ptr<TFC_Client, decltype(&TFC_Destroy)>
        owner(client, &TFC_Destroy);
    check(TFC_Connect(client, server, 500, nullptr, &reply));
    if (useRecipe) {
        check(TFC_SetMeasurementRecipe(client, setupName, &reply));
    } else {
        // Requires a server supporting filmstack-only calculation.
        check(TFC_SetFilmstack(client, setupName, &reply));
        // Keep suitable server conditions, or set them explicitly here.
    }
    check(TFC_Calculate(client, pairs, count, &reply));
    const uint64_t acceptedId = reply.data_id;
    auto deadline = std::chrono::steady_clock::now()
                  + std::chrono::seconds(30); // Application choice.
    TFC_Event event{};
    while (std::chrono::steady_clock::now() < deadline) {
        const int rc = TFC_PollEvent(client, &event);
        if (rc < 0) throw std::runtime_error("Invalid event poll");
        if (rc == 0) {
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
            continue;
        }
        if (event.dropped_events)
            throw std::runtime_error("Events lost; reconcile results");
        if (event.type == TFC_DATA_ERROR)
            std::fprintf(stderr, "%s\n", event.message);
        if (event.type != TFC_RESULT || event.data_id != acceptedId) continue;
        if (event.exception) throw std::runtime_error("Calculation failed");
        std::printf("Warning=%u high_mse=%u outlier=%u\n",
            unsigned(event.warning), unsigned(event.high_mse),
            unsigned(event.outlier));
        if (event.has_mse) std::printf("MSE=%g\n", double(event.mse));
        for (uint32_t i = 0; i < event.parameter_count; ++i)
            std::printf("%s=%g\n", event.parameters[i].name,
                        double(event.parameters[i].value));
        return; // owner destroys the client after all calls have returned.
    }
    throw std::runtime_error("No result within 30s; do not auto-resubmit");
}
```