



Ellisys Analyzer Python SDK Guide

Version 1.0, June 8, 2026

Table of Contents

1. Introduction	1
2. Installing the Remote Control plugin	1
3. Installing the Python SDK	3
4. Getting started	3
5. Key concepts and design	3
5.1. The Analyzer and the connection	3
5.2. Overviews, scopes and handles	4
5.3. Cursors: never read everything by accident	4
5.4. The growth guard (loading and recording)	4
5.5. Searching	5
5.6. Product facets	5
5.7. Typed exports	5
5.8. Errors	5
6. How-to recipes	6
6.1. Extract a whole trace to CSV (constant memory)	6
6.2. Wait for a condition during a live capture	6
6.3. Annotate items found by a search	6
6.4. Build a table for pandas	6
6.5. Survey an unknown trace	6
7. Overriding Connection Properties	7
7.1. By file	7
7.2. By command line	7
7.3. Priority order	7
8. Running the provided samples	7
8.1. C# / .net sample	7
8.2. Python sample	7
8.3. Using a different language or platform	8
9. Overview queries	8
9.1. Syntax	8
9.2. Examples	8
9.3. Behavior notes (important for automation)	9
10. Using with AI agents	9
10.1. Installing the agent skill	10
10.2. The MCP server	10
10.3. Choosing between the skill and the MCP server	11
11. API reference	12
11.1. Connecting and the Analyzer	12
11.1.1. <code>connect(host='localhost', port=12345, *, message_size_max_mb=256)</code>	12

11.1.2. Analyzer	12
11.2. Overview navigation	17
11.2.1. ScopeClosed	17
11.2.2. BoundExceeded	17
11.2.3. OverviewIncomplete	17
11.2.4. ItemField	18
11.2.5. XmlFilter	18
11.2.6. ItemRecord	18
11.2.7. OverviewItem	18
11.2.8. OverviewItemList	20
11.2.9. Page	21
11.2.10. ChildCursor	21
11.2.11. SearchCursor	23
11.2.12. OverviewScope	24
11.3. Bluetooth facet	25
11.3.1. BluetoothChannelSummary	25
11.3.2. BluetoothChannelStat	26
11.3.3. BluetoothSpectrumSample	26
11.3.4. BluetoothSpectrumRange	26
11.3.5. format_bluetooth_bdaddr(addr: int) → str	27
11.3.6. BluetoothPacketLossMode	27
11.3.7. BluetoothExportModes	27
11.3.8. BluetoothAudioOptions	27
11.3.9. BluetoothChannelsOptions	28
11.3.10. BluetoothAirtimeOptions	28
11.3.11. BluetoothFacet	28
11.4. Wi-Fi facet	30
11.4.1. WifiFacet	30
11.5. WPAN / 802.15.4 facet	30
11.5.1. WpanFacet	30
11.6. USB 3.0 facet	31
11.6.1. Usb30Facet	31
11.7. Exports	32
11.7.1. ExportOptionError	32
11.7.2. ExportMode	32
11.7.3. ExportModes	32
11.7.4. render_option_value(value: object) → str None	32
11.7.5. ExportOptions	32
11.7.6. FilteredTraceOptions	33

11.7.7. ThroughputOptions	33
11.8. Markers	33
11.8.1. MarkerColor	33
11.8.2. Marker	33
11.9. Session, trace files and info	34
11.9.1. MessageSeverity	34
11.9.2. AppInfo	34
11.9.3. RecordingStatus	34
11.9.4. RunningTask	34
11.9.5. TraceFileInfo	35
11.10. Logic signals	35
11.10.1. LogicSignalTransitionType	35
11.10.2. LogicSignalTransition	35
11.11. Errors	36
11.11.1. RemoteControlError	36
11.11.2. ConnectionFailed	36
11.11.3. OperationError	36
11.11.4. NotAvailable	36
11.11.5. LoadTimeout	36
Revisions History	36
Copyright and Intellectual Property	36

1. Introduction

`ellisys_analysis_automation` is a Pythonic SDK for the Ellisys analyzer Remote Control automation API. The automation API itself is based on the ZeroC ICE library (see the [Analyzer Remote Control User Guide](#) for the raw, language-neutral API): the Ellisys analysis software is the server and your script is the client.

The SDK wraps that API so you write idiomatic Python instead of dealing with Ice directly:

- `connect()` returns an `Analyzer` you use as a context manager — it hides all Ice setup (proxy creation, the mandatory encoding, the checked cast).
- The Overview is navigated through **bounded cursors**, so a trace with millions of items is safe to traverse by construction — you can never accidentally pull everything into memory.
- Product-specific operations live on **facets** (`analyzer.bluetooth`, `.wifi`, `.wpan`, `.usb30`); exports are typed; times are plain integers (picoseconds) or `datetime.timedelta`; and **no Ice type ever crosses the SDK surface**.
- Errors are a small, catchable hierarchy rooted at `RemoteControlError`.

In practice the SDK covers every non-deprecated operation of the API.



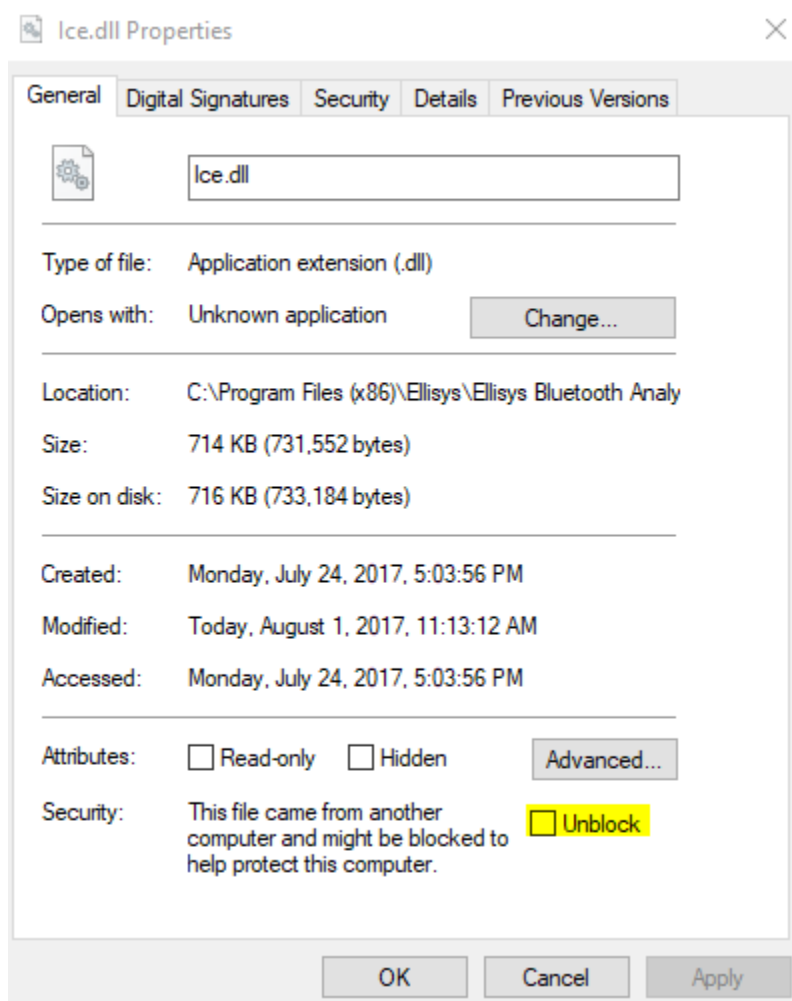
This guide is the recommended entry point for **Python** automation. If you are working in C++ or .NET, or want the underlying operation-level reference, see the [Analyzer Remote Control User Guide](#) and the product manuals.

2. Installing the Remote Control plugin

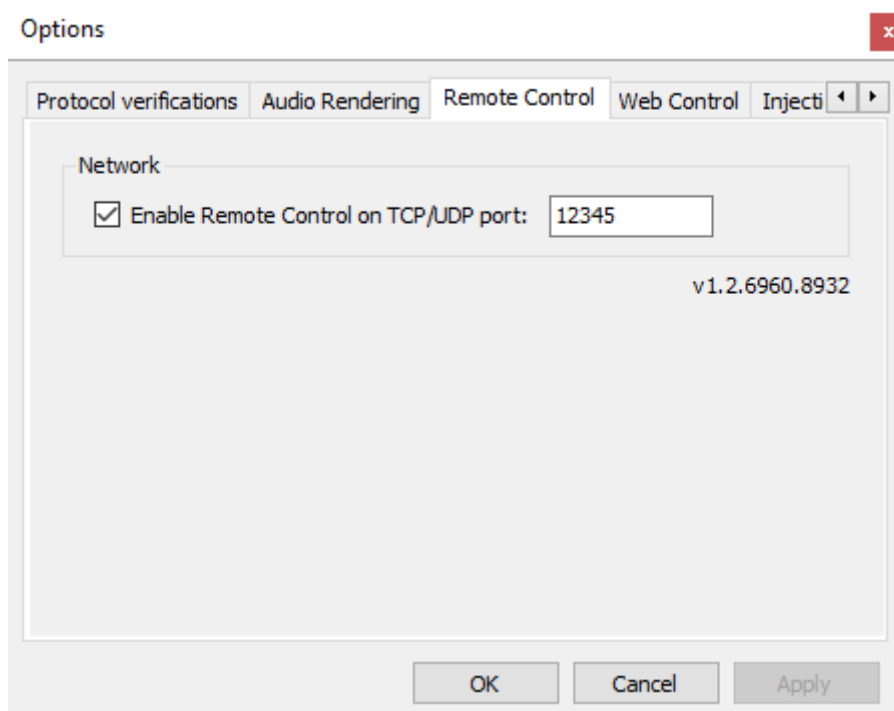
The remote control plugin is available on Ellisys website and is linked from the user manual of the analysis software. Please follow these steps to install the plugin:

1. Install the latest analysis software release.
2. Copy the `Binary\RemoteControl` folder and its content to the installation directory, typically `C:\Program Files (x86)\Ellisys\...Analyzer\RemoteControl`.

Please note that Windows tends to block files downloaded from the Internet. Make sure to unblock them in the file Properties dialog, as shown below:



3. Start the analysis application and go to menu "Tools", then "Options...". If step 2 above was successful you should see a panel named "Remote Control".



4. Check the box "Enable Remote Control on TCP/UDP port", and optionally change the port according to your network parameters. The software is now ready to accept incoming commands.

The port can be overridden by the command line with `/remote_control_port=54321`.

3. Installing the Python SDK

The SDK targets CPython 3.12-3.14 and depends on `zeroc-ice` (3.8.x). Install it from the source tree in editable mode:

```
python -m pip install -e Sdk/Analysis/Python
```

This pulls in `zeroc-ice` and registers the `ellisys_analysis_automation` package. The SDK ships pre-generated Ice bindings and a `py.typed` marker, so editors and type-checkers see full type information with no extra steps.

4. Getting started

Connect to a running analyzer (with the Remote Control plug-in enabled), load a trace, and read the first few Overview items:

```
import ellisys_analysis_automation as ea

with ea.connect("localhost", 12345) as analyzer:
    analyzer.load(r"C:\traces\capture.btt")          # blocking by default; path is on the
    analyzer

    with analyzer.overview("BR/EDR Overview") as ov:
        for item in ov.root.cursor().stream():      # bounded, batched, page-released
            print(item.time, item.description)
            if item.time_ps > 5_000_000_000_000:    # 5 s, in picoseconds
                break
```

The `with` blocks are important: leaving the `analyzer` block disconnects, and leaving the `overview` block releases all of that overview's server-side item handles.

5. Key concepts and design

This section is the mental model. The internal rationale (handle generations, the global release primitive, the `slice2py` mapping) lives in the SDK's `docs/` design notes; here we cover only what a user needs to use the SDK well.

5.1. The Analyzer and the connection

`connect()` returns an `Analyzer`. It is a context manager; use `with` (or call `close()`). Every call is synchronous and blocks until the analyzer replies. Operations that take time (loading a trace, exporting) block accordingly; `load()` waits for completion by default and raises `LoadTimeout` if it exceeds the timeout.

5.2. Overviews, scopes and handles

An **Overview** is a tree of protocol items. You open one as a scope:

```
print(analyzer.overviews())           # available overview names
with analyzer.overview("BR/EDR Overview") as ov:
    print(ov.root.child_count)        # a cheap count, even at millions of items
```

The server addresses items by opaque handles that live until released, and the **only** release primitive is global. So the SDK ties handle lifetime to the `OverviewScope`: when the `with` block exits, every handle minted inside it is freed. Touching an item after its scope closed raises `ScopeClosed` **before** any network call.

5.3. Cursors: never read everything by accident

You never get a plain list of children — that could be millions. Instead you traverse through a **bounded cursor**:

```
cur = ov.root.cursor()                # a ChildCursor; page_size defaults to 256
for item in cur.stream():              # one page at a time, under the hood
    ...

page = ov.root.cursor().page()         # a single bounded Page (<= page_size)
first50 = ov.root.cursor().take(50)    # explicit, bounded; raises BoundExceeded past 100k
slice_ = ov.root.cursor()[1000:1050]  # finite slice; open-ended slices are refused
```

- `stream()` yields live `OverviewItem` objects, paging transparently. By default each page's handles are released at the page boundary, so a full scan keeps server memory at $O(\text{page_size})$. The trade-off: do not hold a live item across a page boundary (it would raise `ScopeClosed`) — snapshot it instead (next bullet), or pass `release_mode="defer"` to `cursor()` to keep handles for the scope.
- `stream_records(*fields)` yields handle-free `ItemRecord` snapshots (description, time, data, XML — whichever fields you ask for). This is the safe full-scan extractor: constant client memory, page-released, and the records never go stale. Ideal for CSV/pandas.
- `take(n)`, `page()`, and `cursor[a:b]` are explicit, bounded reads; `materialize()` and `to_table()` pull everything (raising `BoundExceeded` past the cap).

5.4. The growth guard (loading and recording)

A trace loads asynchronously and a live capture grows the Overview, so its item count is a moving target. A full traversal that snapshotted the count once would stop short and silently drop later items. To prevent that, `stream`, `stream_records`, `walk`, `iter_children` and `materialize` **raise `OverviewIncomplete`** while a trace is loading or recording — unless you say how to handle it:

```
# Wait for a complete trace, then traverse (the simplest, safest path):
analyzer.load(path)                    # load(wait=True) blocks until fully loaded
for item in ov.root.cursor().stream():
    ...

# Or follow the growth, tailing items as they arrive (e.g. during a live capture):
for item in ov.root.cursor().stream(follow=True):
    if matches_condition(item):
        break                          # stop when your condition is met

# Or take an explicit snapshot of what exists right now:
for item in ov.root.cursor().stream(snapshot=True):
```


...

Bounded reads (`take`, `page`, `cursor[a:b]`) are exempt — they are explicitly partial by request, so nothing is lost silently.

5.5. Searching

Search is space-windowed, depth-bounded, capped, and observable — bounded like everything else:

```
sc = ov.search_cursor(description=["*Paging*"], max_depth=8, result_cap=50_000)
for hit in sc.stream():
    hit.add_marker("found", color=ea.MarkerColor.RED)
print(sc.found, "of", sc.scanned, "scanned")
```

5.6. Product facets

Product-specific operations are grouped on facets, reached as properties on the analyzer. They attach lazily and raise `NotAvailable` if the connected analyzer is not that product:

```
summaries = analyzer.bluetooth.channel_summaries() # per-RF-channel tallies
analyzer.wifi.add_wifi_key_by_ap_ssid("HomeNet", "secret")
analyzer.wpan.add_zigbee_network_key(0x1234, key16)
analyzer.usb30.connect_link()
```

5.7. Typed exports

Export is one server operation selected by a mode name; the SDK gives you typed wrappers and typed option carriers, and renders the option values to the wire for you. Output paths are on the **analyzer machine**.

```
import datetime
from ellisys_analysis_automation import ExportModes, BluetoothExportModes,
BluetoothAudioOptions, BluetoothPacketLossMode

analyzer.export_filtered_trace_time_range(
    r"C:\out\first5s.btt", max_duration_ps=datetime.timedelta(seconds=5))
analyzer.export_throughput(r"C:\out\tput.csv")

# Any mode via the generic method + a typed carrier:
analyzer.export(r"C:\out\audio", BluetoothExportModes.AUDIO,

options=BluetoothAudioOptions(packet_loss=BluetoothPacketLossMode.SILENCE_SMOOTH))

# Inspect the exact wire options before committing a long export:
print(analyzer.export_dry_run(r"C:\out\x", ExportModes.THROUGHPUT))
```

5.8. Errors

All SDK errors derive from `RemoteControlError`:

- `ConnectionFailed` — transport/connection problems (or the endpoint is not an analyzer).
- `OperationError` — the analyzer rejected the operation; subclasses: `LoadTimeout`, `ScopeClosed`, `BoundExceeded`, `OverviewIncomplete`.
- `NotAvailable` — a facet/feature is not present on this analyzer.

- `ExportOptionError` (a `ValueError`) — a bad export option, caught client-side.

6. How-to recipes

6.1. Extract a whole trace to CSV (constant memory)

```
import csv
from ellisys_analysis_automation import ItemField

with analyzer.overview("BR/EDR Overview") as ov, open("out.csv", "w", newline="") as f:
    w = csv.writer(f)
    for rec in ov.root.cursor().stream_records(ItemField.TIME, ItemField.DESRIPTION):
        w.writerow([rec.time_seconds, rec.description])
```

6.2. Wait for a condition during a live capture

```
with analyzer.recording(r"C:\traces\live.btt"):      # start; saved when the block exits
    with analyzer.overview("BR/EDR Overview") as ov:
        for item in ov.root.cursor().stream(follow=True):
            if "Disconnect" in item.description:
                break
recording                                           # got what we waited for; stop
```

6.3. Annotate items found by a search

```
with analyzer.overview("BR/EDR Overview") as ov:
    for hit in ov.search_cursor(description=["*Error*"]).stream():
        hit.add_marker("error here", color=ea.MarkerColor.RED)
```

6.4. Build a table for pandas

```
with analyzer.overview("BR/EDR Overview") as ov:
    rows = ov.root.cursor().to_table("Status", "Opcode", cap=5000)    # list[dict],
    raises past cap
    # import pandas; pandas.DataFrame(rows)
```

6.5. Survey an unknown trace

Protocol layers are per-protocol *views* of an overview; switching is instant (no re-processing). Counting root items across overviews and layers fingerprints what traffic a trace contains:

```
with ea.connect() as analyzer:
    for name in analyzer.overviews():
        with analyzer.overview(name) as ov:
            if not ov.root.child_count:
                continue
            print(f"{name}: {ov.root.child_count} items")
            original = ov.protocol_layer
            for layer in ov.available_protocol_layers():
                ov.set_protocol_layer(layer)
                print(f"    {layer}: {ov.root.child_count}")
            ov.set_protocol_layer(original)
            # no traffic in this overview
            # instant; invalidates handles
```

The packaged version of this (plus vocabulary, field-name and device discovery) is `ellisys_analysis_automation.discovery.explore(analyzer)` — see the API reference.

7. Overriding Connection Properties

7.1. By file

ZeroC ICE server offers plenty of parameters to optimize different applications. It is possible to override the default properties by placing a file named `Ice.props` in the `RemoteControl` folder. The file shall be of this format: <https://doc.zeroc.com/display/Ice35/Configuration+File+Syntax>

7.2. By command line

It is also possible to override properties with the command line. Any ICE property can be set on the command line, such as `/Ice.ThreadPool.Server.SizeMax=8`.

7.3. Priority order

The priority is given as follow:

1. Defaults in the application
2. File `Ice.props`
3. Command line

8. Running the provided samples

8.1. C# / .net sample

1. The Samples folder contains a solution compatible with Visual Studio 2017 or higher.
2. Follow these steps to run the sample:
 - a. Start the analysis software previously configured in the above section. Please ensure an analyzer unit is connected to this computer.
 - b. Compile and execute the provided sample from Visual Studio.
 - c. The analysis software should start recording and will wait for a key before stopping the recording.
3. The provided sample can be used as starting point for custom control software.

8.2. Python sample

1. Install the latest Python environment from <https://www.python.org/downloads/>
2. Install the sample's dependencies with the command: `python -m pip install -r requirements.txt`
3. Follow these steps to run the sample:
 - a. Start the analysis software previously configured in the above section. Please ensure an analyzer unit is connected to this computer.
 - b. Run the sample with the command: `python sample.py`

- c. The analysis software should start recording and will wait for a key before stopping the recording.
4. The provided sample can be used as starting point for custom control software.

8.3. Using a different language or platform

The sample is provided with language-agnostic definition of the Ellisys API. The ICE files can be converted to the various supporting languages such as Ruby, JavaScript, and more, with a tool named slice provided by ZeroC. Please consult ZeroC documentation for creating a wrapper for your language and platform of choice:

<https://doc.zeroc.com/ice/latest/the-slice-language/slice-compilation>

9. Overview queries

A query filters the overview to the lines matching criteria on **any column of the overview or any field of the Details view** — the filtering runs inside the analyzer, so it is the most efficient way to narrow millions of items down to the lines of interest before traversing them. Queries are the same feature as the application's query toolbar; through the API they are set with `SetOverviewQuery` (SDKs: Python `overview.set_query(...)`, C# `overview.SetQuery(...)`).

9.1. Syntax

```
Field or column name = [!]value[, value, ...], Another field = [!]value[, ...]
```

- A **term** is `Field = value[, value, ...]`. Values separated by commas are alternatives: the field must match **at least one** of them. An exclamation mark before a value creates a NOT condition. Terms separated by commas must all match; terms can also be put in parentheses and combined with `&&` (and) / `||` (or) instead.
- **Comparators**: `=`, `!=`, `<`, `>`, `<=`, `>=`.
- **Field and column names** are written as displayed, unquoted — multi-word names work (`RF Channel Number >= 40`). Fields that appear several times can be indexed: `Data[1]` is the second `Data` field.
- **Values**:
 - Text in double quotes; matches any run of characters: `"Text"`, `"error"`.
 - Numbers: decimal `123`, hex `0xABCD`, binary `0b010101`; inclusive ranges `7..10`; computations using operators and other fields (`4+5`, `0x0F << 2`).
 - Data patterns: `0x[A1 B2 C3]□□` bytes in hex, `##` matches any single byte, a trailing `□` matches any remaining bytes. Use a leaf data field (e.g. `Raw Data`), not a grouping field.
 - Regular expressions: `Regex("^[hc]at")`.
- **Functions**: `ByteAt(Field, n)` — the field's `n`-th byte (0-based) as a number.

9.2. Examples

Query	Matches lines where
<code>Item = "Text"</code>	the Item column starts with <code>Text</code>
<code>Item = !"Text"</code>	the Item column does not contain <code>Text</code>

Query	Matches lines where
Status = "OK"	the Status column is exactly OK
Foo = 1, 0x03, 7..10	Foo is 1, 3, 7, 8, 9 or 10
Foo >= 0x0F << 2, 4+5 && (Bar = 1 Status != "OK")	Foo is at least 9, and either Bar is 1 or Status is not OK
Item = "AVDTP*" && Status = "OK"	the Item starts with AVDTP and the Status is OK
Payload = 0x[0A ## 0C *]	the Payload's 1st byte is 0x0A and its 3rd byte is 0x0C
ByteAt(Data[1], 3) = 0	the second Data field's 4th byte is 0
Item = RegEx("^([hc]at)")	the Item starts with hat or cat

9.3. Behavior notes (important for automation)

- **A line matches when any of its field occurrences match.** Overview lines summarize the items beneath them, so a field can occur several times per line with different values (a line spanning several packets has several RF Channel Number values, and matches both = 0..39 and >= 40 if its packets span both ranges).
- **The query is applied asynchronously.** SetOverviewQuery returns at once and the overview re-filters in the background, reported as a running task (e.g. "Filtering data" in GetRunningTasks — which can take a moment to appear) while IsLoading stays false; existing item handles are invalidated, and counts read in the meantime may reflect the previous state. Raw-API users should poll GetRunningTasks until the analyzer stays idle. The SDKs handle this: set_query / SetQuery block until the re-filtering finishes by default (pass wait=False / wait: false to return immediately and settle later with wait_until_idle() / WaitUntilIdle()).
- **Syntax errors are reported, unknown names are not.** A malformed query (e.g. unbalanced parentheses, missing value) fails the call with Invalid query; a misspelled field or column name is accepted and simply matches **nothing** (0 lines). If a query unexpectedly returns 0, verify the field name against the overview columns / Details view of a matching item first.
- An empty query ("") clears the filter and restores the full overview.

10. Using with AI agents

The Remote Control distribution is designed to be driven by AI coding agents as well as by humans. The release archive includes three layers of agent-oriented material:

- **Archive index** — llms.txt at the archive root: a Markdown link list describing what the archive contains and where everything is. An agent pointed at the unzipped archive discovers the rest from there.
- **SDK agent guides** — Sdk/Analysis/Python/ellisys_analysis_automation/llms.txt and Sdk/Analysis/CSharp/Ellisys.Analysis.Automation/llms.txt: dense, authoritative API guides with the rules the bounded API enforces and canonical code recipes. They are bundled inside the SDK packages, so they are also available wherever the SDK is installed.
- **Agent skill** — Skill/ellisys-analysis/: a ready-to-install skill for Claude Code packaging the complete workflow: SKILL.md (the mental model, a starter recipe and a troubleshooting table), scripts/probe.py (connectivity diagnosis with actionable hints) and scripts/setup_env.py (environment bootstrap).

10.1. Installing the agent skill

To enable the skill for a given project, copy the `Skill/ellisys-analysis` folder into the project's `.claude/skills/` directory:

```
<project>/
  .claude/
    skills/
      ellisys-analysis/
        SKILL.md
        scripts/           (probe.py, setup_env.py, explore.py)
        sdk/python/        (the bundled Python SDK source)
        reference/         (the Python SDK guide, Markdown)
```

To enable it for all projects of the current user instead, copy the same folder to `~/.claude/skills/` (`%USERPROFILE%\claude\skills\` on Windows). Claude Code activates the skill automatically whenever a task involves controlling an Ellisys analyzer; no further configuration is needed.

The skill folder is **self-contained**: it bundles the Python SDK source (`sdk/python/`) and the SDK guide (`reference/`), so it keeps working when copied on its own, away from this archive. Network access to PyPI is only needed for the `zeroc-ice` dependency at environment setup.

10.2. The MCP server

For AI hosts without shell access — chat applications such as Claude Desktop — Ellisys provides an MCP (Model Context Protocol) server, `ellisys-analysis-mcp`, distributed on PyPI. It exposes four tools: `analysis_guide` (the SDK agent guide and canonical recipes), `analysis_status` (connectivity diagnosis, application/trace state, markers), `analysis_explore` (the discovery pass: overview/protocol-layer survey, vocabulary, devices, anchors) and `analysis_run_python` (a persistent Python session with the SDK and a connected analyzer preloaded — all actions and analysis happen here). The session includes three helpers: `install("pkg", ...)` adds extra libraries (pandas, matplotlib, ...) on demand, `show(figure)` returns a matplotlib figure to the chat as an image, and `attach(path)` returns a small file (e.g. a CSV export; up to 4 MB) with the result. To have matplotlib and pandas preinstalled instead, configure the package with the `analysis` extra: `uvx "ellisys-analysis-mcp[analysis]@latest"`.

The only prerequisite is `uv`, a single small executable that provisions Python and the packages automatically — no Python installation is required. Install it once:

- **Windows:** `winget install astral-sh.uv`
- **macOS:** `brew install uv`
- **Linux (and macOS without Homebrew):** `curl -LsSf https://astral.sh/uv/install.sh | sh`

Then verify with `uvx --version` in a **new** terminal — the installer updates the `PATH` for new processes only — and (re)start the MCP host application afterwards so it sees the updated `PATH`.

The analyzer application must be running with the Remote Control plugin enabled; the server connects to `localhost:12345` by default (override with `--host/--port` arguments or the `ELLISYS_HOST/ELLISYS_PORT` environment variables).

Claude Code (terminal):

```
claude mcp add ellisys-analysis -- uvx ellisys-analysis-mcp@latest
```

Claude Desktop — add to `claude_desktop_config.json` (Settings > Developer > Edit Config):

```
{
  "mcpServers": {
    "ellisys-analysis": { "command": "uvx", "args": ["ellisys-analysis-mcp@latest"] }
  }
}
```

The `@latest` suffix makes `uvx` check for the newest release every time the server starts, so updating is just a restart of the host application. Without a suffix, `uvx` keeps reusing the first version it resolved until its cache is cleared manually (`uv cache clean ellisys-analysis-mcp`); a fixed version can be pinned with `ellisys-analysis-mcp@0.1.1`.



If the host reports the server as **failed** (often error `-32000`), the usual cause is that `uvx` is not on the `PATH` of the host process: verify `uvx --version` works in a **new** terminal, and restart the host application after installing `uv` (it inherits the `PATH` from its launch). Alternatively, configure the full path to `uvx` instead of the bare command.

Other MCP hosts (Cursor, VS Code, ChatGPT desktop, ...): register a custom/local MCP server with the command `uvx` and the argument `ellisys-analysis-mcp@latest` in the host's MCP configuration. Hosts that only accept **remote** (HTTP) MCP servers are not yet supported (a streamable-HTTP transport is planned).

10.3. Choosing between the skill and the MCP server

Both deliver the same capability — they share the SDK, its agent guide and its discovery module — so the choice is purely about the host:

- **Use the skill** with coding agents that have shell and filesystem access (Claude Code and similar). The agent works in its own environment: it can install extra packages alongside the SDK (`pandas`, `matplotlib`, ...), produce files in your project (CSV analyses, plots, reusable scripts) and integrate the analyzer into larger codebases and CI. This is the most capable option for engineering work.
- **Use the MCP server** with chat applications and other hosts that cannot run scripts (Claude Desktop, ChatGPT desktop, ...). Code runs inside the server's session and results come back as text — ideal for interactive exploration and Q&A about a trace.
- Hosts that support both (e.g. Claude Code) can use either; the skill offers more headroom there.

The skill is plain Markdown plus two small Python scripts, so agents other than Claude Code can use `SKILL.md` directly as instructions; the scripts run with any Python 3.10+ on Windows, Linux or macOS.



The agent (the client side) can run on any platform, and the analyzer application runs on Windows natively and on Linux and macOS through the Ellisys-provided runtime — the whole chain is multiplatform. File paths passed to load, save and export operations are resolved by the analyzer application on its machine and must be valid for that installation.

11. API reference

The reference below is generated from the SDK's docstrings; edit those and run `python make.py build` to regenerate.

11.1. Connecting and the Analyzer

11.1.1. `connect(host='localhost', port=12345, *, message_size_max_mb=256)`

Connect to a running Ellisys analyzer that has Remote Control enabled.

Hides all ZeroC Ice setup: it loads the pre-generated bindings, forces the mandatory Ice encoding 1.0, raises the message-size limit, builds the proxy and performs the checked cast. Returns an `Analyzer`; use it as a context manager or call `close()`.

Raises `ConnectionFailed` if the analyzer cannot be reached or the endpoint is not an analyzer.

11.1.2. `Analyzer`

A connected Ellisys analyzer. Obtain one from `connect`.

Supports the context-manager protocol; leaving the `with` block disconnects.

`is_recording` (property)

`True` while a capture is in progress.

`start_recording()`

Start a capture (equivalent to Record > Start in the GUI).

`stop_and_save(filename, *, overwrite=False)`

Stop the capture and save the trace to **filename**.

The path is interpreted on the **analyzer machine**, not the client running this script. Set `overwrite=True` to replace an existing file.

`abort_recording()`

Stop the capture and discard the trace (nothing is saved).

`recording(save_to, *, overwrite=False)`

Record for the duration of the `with` block, then stop and save to **save_to**.

On a normal exit the trace is saved. If the body raises, the trace is still saved (best effort) and the original exception propagates — a capture is rarely worth losing because the script that triggered it failed.

overviews()

Names of the available overviews (→ `GetAvailableOverviews`). No scope needed.

active_overview()

Name of the currently selected overview, or `None` (→ `GetSelectedOverview`).

overview(name=None, *, query=None, protocol_layer=None, restore_active=True)

Open a navigation scope on **name** (or the already-active overview if `None`).

Mirrors `recording`: on entry it snapshots the active overview, selects **name**, and optionally applies **query** / **protocol_layer**; it yields an `OverviewScope` for navigating the tree. On exit it frees all item handles once (`ReleaseAllOverviewItemHandles`) and, unless `restore_active=False`, re-selects the previously active overview.

Only one scope may be open at a time — handle release is global, so a nested scope would free this one's handles. Opening a second raises `OperationError`.

add_marker_at_time(time_ps, text, *, color=MarkerColor.YELLOW)

Add a marker at **time_ps** picoseconds (→ `AddMarkerAtTime`).

color is a `MarkerColor` (or its int value).

add_marker_on_selected_overview_item(text, *, color=MarkerColor.YELLOW)

Add a marker on the currently selected overview item (→ `AddMarkerOnSelectedOverviewItem`).

markers()

Every marker in the trace as handle-free `Marker`'s (→ `GetMarkers`).

export(output, mode, *, options=None)

Run any export mode (→ `Export`). Generic primary method AND escape hatch.

output is a path on the **analyzer machine** (like `stop_and_save`). **mode** is the server `exportName` — an `ExportMode` member or any raw string (including modes newer than this SDK). **options** is a typed carrier (e.g. `BluetoothAudioOptions`) or a raw `{OptionName: value}` mapping keyed by the EXACT server tokens (PascalCase). Values may be `int`, `bool`, `str`, `datetime.timedelta`, or an `Enum`; `None` values are omitted. A product mode used on the wrong analyzer raises `OperationError`.

export_dry_run(output, mode, *, options=None)

Render the exact `(OptionName, OptionValue)` wire pairs WITHOUT sending (offline).

Calls `options.validate()` if present, then returns what `export` would send. Makes no proxy call — useful for tests and for inspecting omit-None / bool / enum / picosecond rendering before committing a long export.

(**output** is accepted for signature symmetry with `export`; it is not part of the option pairs.)

export_filtered_trace_time_range(output, *, start_time_ps=None, max_size_bytes=None, max_duration_ps=None, max_items=None)

Export a filtered trace by time range (BASE; any analyzer).

*_ps accept int picoseconds or a `datetime.timedelta`; `max_size_bytes` is a byte count. `None` omits the option (server defaults: StartTime 0, the rest = full trace).

export_filtered_trace_active_overview(output, *, start_time_ps=None, max_size_bytes=None, max_duration_ps=None, max_items=None)

Export a filtered trace by the active overview (BASE). Same options as the time-range form.

export_throughput(output, *, range_start_time_ps=None, range_end_time_ps=None)

Export throughput over a time range (BASE). `None` → server default (trace start/end).

app_info()

Information about the remote application (→ `GetAppInfo`).

exit_app()

Ask the remote application to exit (→ `ExitApp`).

recording_status()

Status of the current recording (→ `GetRecordingStatus`).

Only valid while a recording is active; the analyzer rejects it otherwise (surfacing as `OperationError`). Gate it on `is_recording`.

get_recording_options(*, relevant_only=False)

The recording options as an opaque string (→ `GetRecordingOptions`).

configure_recording_options(options)

Apply recording options from an opaque string (→ `ConfigureRecordingOptions`).

data_sources()

Available data source unique IDs (→ `GetAvailableDataSources`).

select_data_source(data_source_unique_id)

Select the data source by its unique ID (→ `SelectDataSource`).

selected_data_source()

The selected data source unique ID, or `None` (→ `GetSelectedDataSource`).

is_loading (property)

`True` while a trace file is loading (→ `IsLoading`).

start_loading(filename)

Begin loading **filename** and return immediately (→ `StartLoading`).

Loading is asynchronous: poll `is_loading` (or use `wait_until_loaded` / `load`) to know when it finishes. **filename** is a path on the analyzer machine.

wait_until_loaded(*, timeout=120.0, poll_interval=0.25)

Block until `is_loading` is `False` (raises `LoadTimeout` on timeout).

Polls every **poll_interval** seconds against a monotonic deadline. `timeout=None` waits forever. **poll_interval** must be `> 0`.

load(filename, *, wait=True, timeout=120.0, poll_interval=0.25)

Load a trace file (→ `StartLoading`); by default blocks until it finishes.

The primary entry point: it calls `start_loading` and then, unless `wait=False`, `wait_until_loaded`, so the next statement can use `trace_file_info`. Raises `LoadTimeout` if loading exceeds **timeout** (`None` waits forever). The path is on the analyzer machine.

trace_file_info()

Information about the loaded trace file (→ `GetTraceFileInfo`).

close_trace_file()

Close the loaded trace file (→ `CloseTraceFile`).

is_modified (property)

`True` if the trace has unsaved changes (→ `IsModified`).

save_changes()

Save changes to the loaded trace file (→ `SaveChanges`).

insert_message(severity, message)

Insert a message in the Message Log (→ `InsertMessage`).

severity is a `MessageSeverity` (or its int value).

running_tasks()

Background tasks currently running (→ `GetRunningTasks`).

wait_until_idle(*, timeout=600.0, poll_interval=0.25, settle=1.0)

Block until no background task is running (polls → `GetRunningTasks`).

Filter changes (overview queries, device filters) re-process the trace in the background, reported as a running task (e.g. "Filtering data") while `is_loading` stays `False` — so this, not `wait_until_loaded`, is the wait that matches them. The background task can take a moment to **appear** after the triggering call, so idleness must persist for **settle** seconds before this returns (`settle=0` returns on the first idle observation).

`timeout=None` waits forever; raises `OperationError` on timeout. **poll_interval** must be `> 0`.

abort_running_task(name)

Abort the running task named **name** (→ `AbortRunningTask`).

get_settings()

The application settings as an opaque `bytes` blob (→ `GetSettings`).

configure_settings(settings)

Apply an opaque settings `bytes` blob (→ `ConfigureSettings`).

cancel_user_interaction()

Cancel a pending user interaction on the analyzer (→ `CancelUserInteraction`).

logic_signals_state(time_ps)

Bitmask of all logic-signal levels at **time_ps** (bit `N` == signal `N`) (→ `GetLogicSignalsState`).

find_logic_signal_transition(from_time_ps, to_time_ps, signals_mask, transition_type=LogicSignalTransitionType.ANY)

Search [**from_time_ps**, **to_time_ps**] for a transition on the masked signals.

signals_mask selects signals by bit (set bit `N` to include signal `N`); **transition_type** is a `LogicSignalTransitionType`. Returns a `LogicSignalTransition` (state bitmask + time). Raises `OperationError` if no matching transition is found. (→ `FindLogicSignalsTransition`)

bluetooth (property)

The Bluetooth-specific facet (→ `BluetoothAnalyzerRemoteControl`), cached per analyzer.

Valid only on a Bluetooth analyzer; raises `NotAvailable` if the connected analyzer is a different product.

wifi (property)

The Wi-Fi facet (→ `EthernetAnalyzerRemoteControl`), cached; `NotAvailable` if not a Wi-Fi analyzer.

wpan (property)

The WPAN facet (→ `WpanAnalyzerRemoteControl`), cached; `NotAvailable` if not a WPAN analyzer.

usb30 (property)

The USB 3.0 facet (→ `Usb30AnalyzerRemoteControl`), cached; `NotAvailable` if not a USB 3.0 analyzer.

close()

Disconnect and release the Ice communicator. Idempotent.

11.2. Overview navigation**11.2.1. ScopeClosed**

A handle was used after its `OverviewScope` released it.

Raised when an item/page is touched after its scope exited, after `OverviewScope.release_handles`, or after a page-streaming cursor released the page that minted it. Subclasses `OperationError`, so existing `except OperationError` handlers still catch it.

11.2.2. BoundExceeded

A bounded read was asked to materialize more than its cap.

Raised by `take(n)` / `cursor[a:b]` / `materialize(cap=)` / `search_result_cap` when the requested or discovered size would exceed the limit, and by constructing a `Page` larger than `HARD_CAP`. Fail-loud by design: there is no silent truncation.

11.2.3. OverviewIncomplete

A full traversal was attempted while the overview is still being populated.

The overview grows while a trace is **loading** or while the analyzer is **recording**, so a plain `stream` / `stream_records` / `walk` / `iter_children` / `materialize` would stop at whatever count happened to exist when it started and silently drop the rest. Rather than truncate quietly, those ops raise this unless the caller says how to handle the growth:

- wait until it is complete first (`analyzer.load(wait=True)` / `wait_until_loaded()`),
- pass `follow=True` to keep traversing as items arrive (live-tail), or
- pass `snapshot=True` to deliberately traverse only what exists right now.

Bounded reads (`take(n)` / `page()` / `cursor[a:b]`) are exempt — they are explicitly partial by request, so nothing is lost silently. Subclasses `OperationError`.

11.2.4. ItemField

Selects which batch operation a reader issues. One member == one plural Ice op.

Members: DESCRIPTION, TIME, DATA, XML

`xml_filtered(*field_names: str) → XmlFilter`

Filtered Details XML (→ `GetOverviewItemsXmlReportFiltered`).

11.2.5. XmlFilter

A request for the filtered Details XML of the given field names.

Fields:

- `field_names: tuple[str, ...]`

11.2.6. ItemRecord

A pure-data, handle-FREE snapshot of one item — safe to keep after the scope closes.

Only the fields you requested are populated; the rest stay `None`. `position` is the item's index within its parent / result set (for traceability) — it is **not** a handle.

Fields:

- `position: int`
- `description: str | None`
- `time_ps: int | None`
- `data: bytes | None`
- `xml: str | None`

time (property) → datetime.timedelta | None

`time_ps` as a `datetime.timedelta` (microsecond resolution), or `None`.

time_seconds (property) → float | None

`time_ps` in seconds as a float, or `None`.

11.2.7. OverviewItem

A lazy, scope-bound wrapper around one opaque handle.

Constructing one is free. Each single-item field property does one Ice call on first access then caches it. For many items, navigate with `cursor` (bounded) or stream with `iter_children` / `walk`.

`_path` is the item's index path from the overview root (e.g. `(3, 7)` == root's child 3, its child 7). It lets a page-releasing cursor re-acquire the item after a global release. Items whose path is unknown (search results, the selected item) carry `None` and cannot be re-seated.

handle (property) → int

The raw server handle — an escape hatch for ops the SDK doesn't wrap yet.

position (property) → int

Index of this item within its parent / result set.

depth (property) → int

Depth relative to where a `walk` started (0 for that level's children).

description (property) → str

Text of the Item column (→ `GetOverviewItemDescription`).

time_ps (property) → int

Timestamp in picoseconds, lossless 64-bit (→ `GetOverviewItemTimeInPicoseconds`).

time (property) → `datetime.timedelta`

Timestamp as a `datetime.timedelta` (microsecond resolution).

data (property) → bytes

Bytes shown in the Raw data view (→ `GetOverviewItemData`).

`xml_report(*, fields: Sequence[str] | None=None) → str`

Details view as XML (→ `GetOverviewItemXmlReport` / ...Filtered).

child_count (property) → int

Number of direct children (→ `GetOverviewItemChildCount`, cached). Cheap even at millions.

`cursor(*, page_size: int=PAGE_DEFAULT, prefetch: Sequence[ItemField] | None=DEFAULT_PREFETCH, release_mode: Literal['page', 'defer']='page') → ChildCursor`

A bounded cursor over this item's children (replaces the old unbounded `children()`).

The cursor yields one `Page` at a time. `page_size` is clamped to `PAGE_MAX`. With `release_mode="page"` (default) each page's handles are freed at the page boundary, so a full scan keeps server memory at $O(\text{page_size})$ — at the cost that a live item must not be held across a page boundary (it raises `ScopeClosed`; use `ChildCursor.stream_records` to keep data). Use `release_mode="defer"` to retain handles (e.g. while holding live items or running concurrent cursors).

`child(index: int) → OverviewItem`

The single child at `index` (→ `GetOverviewItemChild`).

iter_children(*, batch_size: int=PAGE_DEFAULT, follow: bool=False, snapshot: bool=False, poll_interval: float=_FOLLOW_POLL) → Iterator[OverviewItem]

Lazily page through direct children ($\text{ceil}(N/\text{batch_size})$ calls, not N).

A true generator for **bounded** exploration: break early and only fetched pages cost round-trips. It does NOT release handles per page (it yields live items), so for a full multi-million-item scan prefer

`ChildCursor.stream_records`, which is page-released.

While the overview is growing (loading/recording) this raises `OverviewIncomplete` unless you pass `follow=True` (live-tail the children as they arrive) or `snapshot=True` (only the children that exist now).

walk(*, max_depth: int | None=None, batch_size: int=PAGE_DEFAULT, follow: bool=False, snapshot: bool=False, poll_interval: float=_FOLLOW_POLL) → Iterator[OverviewItem]

Lazy depth-first walk of descendants; sets each item's `depth`.

Yields this item's children at depth 0, their children at depth 1, etc. **max_depth** limits the levels (`None` = unbounded). Built on `iter_children` (no per-page release, since it holds ancestors while descending) — intended for bounded exploration with early `break`, not for materializing a whole huge tree.

On a growing overview this raises `OverviewIncomplete` unless you pass `snapshot=True` (walk what exists now) or `follow=True` (wait until loading / recording stops, then walk the complete tree — live-tailing a recursive walk is not well-defined, so `follow` here means "let it finish first").

select() → None

Highlight this item in the analyzer GUI (→ `SelectOverviewItem`).

add_marker(text: str, *, color: MarkerColor=MarkerColor.YELLOW) → None

Select this item and drop a marker on it (→ `SelectOverviewItem` + `AddMarkerOnSelectedOverviewItem`).

record(*fields: ItemField | XmlFilter) → ItemRecord

Snapshot this item to a handle-free `ItemRecord` (single-item ops).

11.2.8. OverviewItemList

An ordered, scope-bound, batch-backed sequence of `OverviewItem` (base of `Page`).

No longer constructed directly by users — a bounded `Page` is what cursors hand out. `len` / indexing / slicing / iteration are pure-local; each batch reader is one bounded round-trip over the list's handles. Construction is capped at `HARD_CAP`.

handles (property) → tuple[int, ...]

The raw handles, in order (escape hatch).

descriptions() → list[str]

Item-column text for every item (→ `GetOverviewItemsDescription`).

times_ps() → list[int]

Timestamps in picoseconds (→ `GetOverviewItemsTimeInPicoseconds`).

times() → list[datetime.timedelta]

Timestamps as timedeltas (microsecond resolution).

data() → list[bytes]

Raw-data-view bytes for every item (→ `GetOverviewItemsData`).

xml_reports(*, fields: Sequence[str] | None=None) → list[str]

Details XML for every item (→ `GetOverviewItemsXmlReport` / ...Filtered).

records(*fields: ItemField | XmlFilter) → list[ItemRecord]

Snapshot the whole list to handle-free records (one batch per requested field).

to_rows(*field_names: str) → list[dict[str, str]]

Filtered Details fields parsed into dicts (for pandas / CSV).

filter(predicate: Callable[[OverviewItem], bool]) → OverviewItemList

A new list of the items matching **predicate** (reuses the same item objects).

11.2.9. Page

One bounded window of children/matches — the only object carrying whole-set readers.

A `Page` is an `OverviewItemList` (so it has the batch readers / slicing / filter) whose length is capped by construction (□ `PAGE_MAX` for a streamed page, □ `HARD_CAP` for a `take/slice` result). It also knows its `position` in the parent and whether more follow (`has_more`).

next_page() → Page | None

Advance the owning cursor and return the next page (None at the end).

11.2.10. ChildCursor

A bounded, page-at-a-time cursor over one item's children. Built by `OverviewItem.cursor`.

Streaming (`stream`, `stream_records`) and manual paging (`page`, `next_page`, `seek`, `reset`) page in windows of `page_size` (clamped to `PAGE_MAX`). Materializing (`take`, `cursor[a:b]`, `materialize`) is bounded and raises `BoundExceeded` past `HARD_CAP`. It is NOT a `Sequence`; `total` is the cheap count, not a fetch of items.

total (property) → int

Number of children (→ `GetOverviewItemChildCount`). Cheap; not a fetch of items.

position (property) → int

Index of the next unread child.

page(size: int | None=None) → Page

The page at the current position (peek; does not advance). `size` clamped to `PAGE_MAX`.

next_page() → Page | None

Advance past the current page (releasing it if `release_mode='page'`); `None` at end.

reset() → None

Release any live page and seek back to the start.

seek(index: int) → None

Release any live page and jump the position to **index**.

stream(*, prefetch: Sequence[ItemField] | None=None, follow: bool=False, snapshot: bool=False, poll_interval: float=_FOLLOW_POLL) → Iterator[OverviewItem]

Yield children one at a time, paging under the hood (batched, bounded, page-released).

The obvious loop on a 10M-child node: $O(\text{page_size})$ handles in flight, ~3 ops/page, early `break` stops fetching. With `release_mode='page'` an item must be consumed within its page — don't stash live items across iterations (use `stream_records`).

While the overview is still growing (loading/recording) this raises `OverviewIncomplete` unless you pass `follow=True` (re-read the count at the end and keep yielding new items as they arrive, until growth stops or you `break`) or `snapshot=True` (traverse only the items that exist now).

stream_records(*fields: ItemField | XmlFilter, page_size: int | None=None, follow: bool=False, snapshot: bool=False, poll_interval: float=_FOLLOW_POLL) → Iterator[ItemRecord]

Yield handle-free `ItemRecord`s, page by page, releasing each page.

The safe full-scan extractor: constant client memory, $O(\text{page_size})$ server handles, batched (one round-trip per field per page). Records survive the per-page release. Defaults to description + time. `follow` / `snapshot` behave as in `stream` (a growing overview otherwise raises `OverviewIncomplete`).

take(n: int) → Page

A Page of the first `min(n, total)` children. Raises if `n > HARD_CAP`.

materialize(*, cap: int=HARD_CAP, fields: Sequence[ItemField | XmlFilter]=DEFAULT_PREFETCH, follow: bool=False, snapshot: bool=False, poll_interval: float=_FOLLOW_POLL) → list[ItemRecord]

All children as handle-free records (page-released). Raises if `total > cap`.

On a growing overview this raises `OverviewIncomplete` unless you pass `snapshot=True` (materialize what exists now) or `follow=True` (wait until loading / recording stops, then materialize the complete set — so do not follow an open-ended live recording here; use `stream` and `break` for that).

to_table(*field_names: str, cap: int=HARD_CAP, follow: bool=False, snapshot: bool=False) → list[dict[str, str]]

All children's filtered Details fields as dicts (for pandas / CSV). Raises if over **cap**.

`follow` / `snapshot` are forwarded to `materialize` (a growing overview otherwise raises `OverviewIncomplete`).

11.2.11. SearchCursor

A bounded cursor over search matches. Built by `OverviewScope.search_cursor`.

`SearchOverviewItems` has no result paging, so this bounds the search SPACE: it windows the base's DIRECT children in steps of `search_fanout`, issuing one search per window and (by default) releasing each window's match handles before the next. `max_depth` is finite by default; `result_cap` raises `BoundExceeded` if total matches exceed it. `scanned` / `found` expose progress.

Residual risk (inherent to the server): a single windowed child whose subtree alone holds millions of matches still returns them in one array; `max_depth` and `result_cap` cap the damage to a loud raise rather than an OOM, but cannot subdivide one subtree.

scanned (property) → int

Direct children of the base searched so far.

found (property) → int

Matches yielded so far.

next_window() → Page | None

The next window's matches as a `Page` (`None` when the search space is exhausted).

stream(*, snapshot: bool=False) → Iterator[OverviewItem]

Yield matches one at a time, windowing the search space (consume within each window).

Like the other full traversals this raises `OverviewIncomplete` while the overview is loading or recording: the searched child count is captured once, so a live search would silently miss later arrivals. Pass `snapshot=True` to search just what exists now.

reset() → None

Release matches and restart the search from the first window.

take(n: int) → Page

A Page of the first `min(n, matches)` matches. Raises if `n > HARD_CAP`.

materialize(*, cap: int=HARD_CAP, fields: Sequence[ItemField | XmlFilter]=DEFAULT_PREFETCH, snapshot: bool=False) → list[ItemRecord]

All matches as handle-free records (window-released). Raises if matches exceed **cap**.

Raises `OverviewIncomplete` while loading or recording; pass `snapshot=True` to materialize just what exists now.

11.2.12. OverviewScope

Owns one server-side handle session for one active overview.

Constructed only by `Analyzer.overview` — never directly. It is the single place that calls `ReleaseAllOverviewItemHandles` (on exit, via `release_handles`, or at a page-streaming cursor's page boundaries). Every item / page / cursor it mints checks the scope is live before any network call.

name (property) → str | None

The overview this scope is bound to.

closed (property) → bool

`True` once the scope has exited.

root (property) → OverviewItem

The active overview's root item (→ `OverviewRootItem`; re-fetched if released).

query (property) → str

The active overview's filter query (→ `GetOverviewQuery`).

set_query(query: str, *, wait: bool=True, timeout: float | None=600.0) → None

Apply a filter query (→ `SetOverviewQuery`). Invalidates existing handles.

Queries filter the overview, inside the analyzer, on any overview column or Details field — e.g. `Item = "AVDTP*" && Status != "OK"`. Comparators = `!= < > □ >=`; comma-separated alternative values; `!` before a value for NOT; `&&/||` and parentheses; numbers (123, 0xABCD, 0b0101, inclusive ranges 7..10); data patterns `0x[A1] (` (= any byte, trailing = any rest); `Regex("...")`; `ByteAt(Field, n)`. The analyzer re-filters the overview in the background (a "Filtering data" running task; `is_loading` stays `False`): by default this call **blocks** until that finishes. Pass `wait=False` to return immediately and settle later via `Analyzer.wait_until_idle`. A malformed query raises `OperationError` ("Invalid query"); an unknown field name silently matches **nothing**. "" clears the filter. See the SDK guide's "Overview queries" section.

protocol_layer (property) → str

The active overview's protocol layer (→ GetSelectedProtocolLayer).

set_protocol_layer(layer: str) → None

Switch the protocol layer (→ SelectProtocolLayer). Invalidates existing handles.

Protocol layers are per-protocol **views** of the overview; switching is instant (no background re-processing, unlike queries/device filters). Iterating `available_protocol_layers` and reading `root.child_count` per layer is the quickest way to fingerprint what traffic a trace contains.

available_protocol_layers() → list[str]

Protocol layers selectable on the active overview (→ GetAvailableProtocolLayers).

search_cursor(*, description: Sequence[str]=(), field_name: Sequence[str]=(), field_value: Sequence[str]=(), within: OverviewItem | None=None, search_fanout: int=DEFAULT_SEARCH_FANOUT, max_depth: int | None=DEFAULT_SEARCH_DEPTH, result_cap: int=DEFAULT_RESULT_CAP, prefetch: Sequence[ItemField] | None=DEFAULT_PREFETCH) → SearchCursor

A bounded cursor over matches under **within** (default: the scope root).

Filters are **glob patterns** (/ ?) or a .NET regex prefixed `regex::`; a bare literal must match the *whole value. `field_name / field_value` filter the Details fields, pairing by index when both are given. `max_depth` defaults to a finite `DEFAULT_SEARCH_DEPTH` (pass `None` for the whole subtree); `result_cap` bounds the total matches before `BoundExceeded`.

selected (property) → OverviewItem | None

The item highlighted in the GUI, or `None` (→ GetSelectedOverviewItem).

select(item: OverviewItem) → None

Highlight **item** in the GUI (→ SelectOverviewItem).

release_handles() → None

Free **all** item handles now without closing the scope (→ ReleaseAllOverviewItemHandles).

Global by nature — there is no per-handle release. Every live item/page minted in this scope raises `ScopeClosed` on next use; already-extracted `ItemRecord`s stay valid. This is also the primitive a page-streaming cursor uses at page boundaries.

11.3. Bluetooth facet

11.3.1. BluetoothChannelSummary

Per-RF-channel packet tallies (→ Slice struct `ChannelSummary`).

Fields:

- `ok: int`
- `retransmitted: int`
- `payload_error: int`
- `header_error: int`
- `not_applicable: int`

errors (property) → int

Header + payload errors.

total (property) → int

Classified packets on this channel (excludes `not_applicable`).

error_rate (property) → float | None

Fraction of classified packets in error, or `None` if the channel had no data.

11.3.2. BluetoothChannelStat

An (RF channel, summary) pair — the explicit-index view of channel summaries.

Fields:

- `rf_channel: int`
- `summary: BluetoothChannelSummary`

11.3.3. BluetoothSpectrumSample

One RSSI sample (→ `GetSpectrumRssi`).

Fields:

- `rssi_dbm: float`
- `start_ps: int`
- `stop_ps: int`

window (property) → datetime.timedelta

Length of the sampling window (`stop_ps - start_ps`) as a `timedelta`.

11.3.4. BluetoothSpectrumRange

A series of RSSI samples over a window (→ `GetSpectrumRssiRange`).

Fields:

- `rssi_dbm: tuple[float, ...]`

- `start_ps: int`
- `stop_ps: int`

window (property) → `datetime.timedelta`

Length of the sampling window (`stop_ps - start_ps`) as a `timedelta`.

mean_dbm (property) → `float` | `None`

Mean RSSI across the samples in dBm, or `None` if there are none.

min_dbm (property) → `float` | `None`

Lowest RSSI sample in dBm, or `None` if there are none.

max_dbm (property) → `float` | `None`

Highest RSSI sample in dBm, or `None` if there are none.

11.3.5. `format_bluetooth_bdaddr(addr: int) → str`

Format a 48-bit BDADDR int as "AA:BB:CC:DD:EE:FF" (a convenience; the API uses ints).

11.3.6. `BluetoothPacketLossMode`

`bluetooth_audio` `PacketLoss` option; each member's value is the exact server token.

Members: `NEGATIVE_DC`, `SILENCE`, `SILENCE_SMOOTH`

11.3.7. `BluetoothExportModes`

Bluetooth export modes (valid only on a Bluetooth analyzer). Enumerate via `ALL`.

11.3.8. `BluetoothAudioOptions`

Options for `bluetooth_audio` (Bluetooth analyzers only).

Fields:

- `add_raw_payload_files: bool` | `None`
- `synchro_buffer_ms: int` | `None`
- `force_isochronous_buffering: bool` | `None`
- `packet_loss: BluetoothPacketLossMode` | `None`

`as_options()` → `dict[str, object]`

The `{server OptionName: value}` mapping for this mode (unset options are dropped).

validate() → None

No range constraints to check.

11.3.9. BluetoothChannelsOptions

Options for `bluetooth_channels` (Bluetooth analyzers only).

Fields:

- `range_start_time_ps`: int | datetime.timedelta | None
- `range_end_time_ps`: int | datetime.timedelta | None

as_options() → dict[str, object]

The {server OptionName: value} mapping for this mode (unset options are dropped).

validate() → None

Validate the range — raises `ExportOptionError` if range end precedes start.

11.3.10. BluetoothAirtimeOptions

Options for `airtime` (Bluetooth analyzers only).

Fields:

- `range_start_time_ps`: int | datetime.timedelta | None
- `range_end_time_ps`: int | datetime.timedelta | None

as_options() → dict[str, object]

The {server OptionName: value} mapping for this mode (unset options are dropped).

validate() → None

Validate the range — raises `ExportOptionError` if range end precedes start.

11.3.11. BluetoothFacet

Bluetooth-only operations on a connected analyzer. Obtain via `analyzer.bluetooth`.

split_trace_and_continue(filename) → None

Save the current trace and keep recording — "Save & Continue" (→ `SplitTraceFileAndContinueRecording`).

filename is on the analyzer machine.

spectrum_rssi(time_ps, rf_channel: int) → `BluetoothSpectrumSample`

RSSI at **time_ps** on **rf_channel** (0-78) (→ `GetSpectrumRssi`).

spectrum_rssi_range(from_time_ps, to_time_ps, rf_channel: int) → BluetoothSpectrumRange

RSSI samples across [from_time_ps, to_time_ps] on rf_channel (→ GetSpectrumRssiRange).

channel_summaries(from_time_ps=0, to_time_ps=None) → list[BluetoothChannelSummary]

Per-RF-channel summaries; list index == RF channel (→ GetChannelsSummary).

Defaults to the whole trace (from_time_ps=0, to_time_ps=None → trace end).

channels(from_time_ps=0, to_time_ps=None) → list[BluetoothChannelStat]

Per-channel summaries as explicit (rf_channel, summary) pairs.

configure_device_filter(mode: DeviceFilterMode, device_addrs: Sequence[int | str]=(), *, wait: bool=True, timeout: float | None=600.0) → None

Set which devices are kept/excluded (→ ConfigureDeviceFilter).

mode is a `DeviceFilterMode`; **device_addrs** are 48-bit BDADDRs — ints or strings as the analyzer displays them ("AA:BB:CC:DD:EE:FF"); ignored for `KEEP_ALL` / `EXCLUDE_BACKGROUND`. The analyzer re-filters in the background: by default this call **blocks** until that finishes (`wait=False` returns immediately; settle later via `Analyzer.wait_until_idle`).

add_link_key(bdaddr1: int | str, bdaddr2: int | str, link_key) → None

Add a 16-byte link key for the **bdaddr1** ↔ **bdaddr2** link (→ AddLinkKey).

Addresses are 48-bit BDADDRs — ints or strings ("AA:BB:CC:DD:EE:FF").

export_audio(output, *, add_raw_payload_files=None, synchro_buffer_ms=None, force_isochronous_buffering=None, packet_loss: 'BluetoothPacketLossMode | None'=None) → None

Export Bluetooth audio via the base `Export` pipeline (mode `bluetooth_audio`).

Builds the `BluetoothAudioOptions` carrier and delegates to `Analyzer.export`. (The legacy dedicated `ExportAudio` op is intentionally not wrapped; this is the supported audio export.)

export_channels(output, *, range_start_time_ps=None, range_end_time_ps=None) → None

Export per-channel statistics (mode `bluetooth_channels`).

export_airtime(output, *, range_start_time_ps=None, range_end_time_ps=None) → None

Export airtime usage (mode `airtime`).

export_mobile_phone_data(output) → None

Export Bluetooth mobile-phone data (mode `bluetooth_mobile_phone_data`; no options).

11.4. Wi-Fi facet

11.4.1. WifiFacet

Wi-Fi-only operations on a connected analyzer. Obtain via `analyzer.wifi`.

add_wifi_key_by_ap_ssid(ap_ssid: str, key: str) → None

Add a Wi-Fi key for the access point with SSID **ap_ssid** (→ `AddWifiKeyByApSsid`).

add_wifi_key_by_ap_mac(ap_mac_addr: int | str, key: str) → None

Add a Wi-Fi key for the access point with BSSID **ap_mac_addr** — a 48-bit int or a string like "AA:BB:CC:DD:EE:FF" (→ `AddWifiKeyByApMacAddr`).

configure_device_filter(mode: DeviceFilterMode, device_addrs: Sequence[int | str]=(), *, wait: bool=True, timeout: float | None=600.0) → None

Set which devices are kept/excluded (→ `ConfigureDeviceFilter`).

mode is a `DeviceFilterMode`; **device_addrs** are 48-bit MACs — ints or strings ("AA:BB:CC:DD:EE:FF"); ignored for `KEEP_ALL` / `EXCLUDE_BACKGROUND`. The analyzer re-filters in the background: by default this call **blocks** until that finishes (`wait=False` returns immediately).

11.5. WPAN / 802.15.4 facet

11.5.1. WpanFacet

WPAN-only operations on a connected analyzer. Obtain via `analyzer.wpan`.

add_thread_master_key(pan_id: int, key128, sequence_counter: int) → None

Add a Thread network master key (16 bytes) for **pan_id** (→ `AddThreadNetworkMasterKey`).

remove_all_thread_master_keys() → None

Remove all Thread network master keys (→ `RemoveThreadNetworkAllMasterKeys`).

remove_thread_master_keys(pan_id: int) → None

Remove all Thread master keys for **pan_id** (→ `RemoveThreadNetworkMasterKeys`).

remove_thread_master_key(pan_id: int, key128) → None

Remove one Thread master key for **pan_id** (→ `RemoveThreadNetworkMasterKey`).

add_zigbee_aps_key(destination_address64, source_address64, key128) → None

Add a Zigbee APS link key (16 bytes) for a device pair (→ `AddZigbeeApsKey`).

remove_all_zigbee_aps_keys() → None

Remove all Zigbee APS link keys (→ RemoveZigbeeApsAllKeys).

remove_zigbee_aps_keys(destination_address64, source_address64) → None

Remove all Zigbee APS link keys for a device pair (→ RemoveZigbeeApsKeys).

remove_zigbee_aps_key(destination_address64, source_address64, key128) → None

Remove one Zigbee APS link key for a device pair (→ RemoveZigbeeApsKey).

add_zigbee_network_key(pan_id: int, key128) → None

Add a Zigbee network key (16 bytes) for **pan_id** (→ AddZigbeeNetworkKey).

remove_all_zigbee_network_keys() → None

Remove all Zigbee network keys (→ RemoveZigbeeNetworkAllKeys).

remove_zigbee_network_keys(pan_id: int) → None

Remove all Zigbee network keys for **pan_id** (→ RemoveZigbeeNetworkKeys).

remove_zigbee_network_key(pan_id: int, key128) → None

Remove one Zigbee network key for **pan_id** (→ RemoveZigbeeNetworkKey).

11.6. USB 3.0 facet

11.6.1. Usb30Facet

USB 3.0-only operations on a connected analyzer. Obtain via `analyzer.usb30`.

connect_link(disconnect_super_speed: bool=False) → None

Connect the analyzer's USB link to the device under test (→ ConnectLink).

disconnect_super_speed keeps only the USB 2.0 link connected when `True`.

disconnect_link() → None

Disconnect the analyzer's USB link from the device under test (→ DisconnectLink).

save_usb20_packets(filename) → None

Save the captured USB 2.0 packets to **filename** on the analyzer (→ SaveUsb20Packets).

save_usb30_symbols(filename, upstream_symbols: bool) → None

Save captured USB 3.0 symbols to **filename** (→ SaveUsb30Symbols).

`upstream_symbols` selects the upstream (`True`) or downstream (`False`) direction.

11.7. Exports

11.7.1. ExportOptionError

A client-side export-option problem (bad value type, failed validation).

Distinct from `OperationError` (server-side, e.g. a wrong-product mode) and `ConnectionFailed`.

11.7.2. ExportMode

A known `exportName` token plus its metadata.

`value` is the exact wire token; `product` is "base" / "bluetooth" / "" (unknown). Equality and hashing are by `value` only, and `str(mode)` is the token, so a mode compares equal to its raw string and methods accept `str` | `ExportMode` — modes newer than this SDK stay reachable by passing the raw token. Use the `ExportModes` (base) and product catalogues (e.g. `BluetoothExportModes`) for the known ones.

Fields:

- `value: str`
- `product: str`
- `description: str`

11.7.3. ExportModes

The base export modes (valid on any analyzer). Enumerate them via `ALL`.

11.7.4. render_option_value(value: object) → str | None

Render ONE typed option value to its exact wire string, or `None` to OMIT it.

The single source of truth for the wire rules: `* None → None` (unset; never sent, server default applies) `* bool → "true" / "false"` (.NET `bool.Parse` form) `* enum.Enum → str(value.value)` (exact token, e.g. "SilenceSmooth") `* timedelta → integer picoseconds as a decimal string` `* int → decimal string` `* str → verbatim`

`bool` is checked before `int` (`bool` subclasses `int`) and `enum` before `int` (an `IntEnum` is an `int`).

11.7.5. ExportOptions

Anything that can supply server-spelled option pairs (every carrier implements it).

`as_options() → dict[str, object]`

Return `{ServerOptionName: typed value}` (the snake→Pascal mapping lives here).

`validate() → None`

Validate the options, raising `ExportOptionError` on a contradictory value (e.g. a time range whose end precedes its start). Carriers without constraints are a no-op.

11.7.6. FilteredTraceOptions

Options for `filtered_trace_time_range` and `filtered_trace_active_overview`.

Fields:

- `start_time_ps: int | datetime.timedelta | None`
- `max_size_bytes: int | None`
- `max_duration_ps: int | datetime.timedelta | None`
- `max_items: int | None`

`as_options()` → `dict[str, object]`

The `{server OptionName: value}` mapping for this mode (unset options are dropped).

`validate()` → `None`

No range constraints to check.

11.7.7. ThroughputOptions

Options for `throughput`.

Fields:

- `range_start_time_ps: int | datetime.timedelta | None`
- `range_end_time_ps: int | datetime.timedelta | None`

`as_options()` → `dict[str, object]`

The `{server OptionName: value}` mapping for this mode (unset options are dropped).

`validate()` → `None`

Validate the options — raises `ExportOptionError` if range end precedes start.

11.8. Markers

11.8.1. MarkerColor

A marker colour (values match the analyzer's `MarkerColor` Slice enum).

Members: `YELLOW`, `BLUE`, `RED`, `GREEN`, `ORANGE`, `PURPLE`

11.8.2. Marker

A marker read from the trace (→ `GetMarkers`). Handle-free; `time_ps` is picoseconds.

Fields:

- `color: MarkerColor`

- `text: str`
- `time_ps: int`

time (property) → datetime.timedelta

`time_ps` as a `datetime.timedelta` (microsecond resolution).

time_seconds (property) → float

`time_ps` in seconds as a float.

11.9. Session, trace files and info

11.9.1. MessageSeverity

Severity for `Analyzer.insert_message` (mirrors the `Slice` enum; 1-based).

Members: `INFO`, `WARNING`, `ERROR`

11.9.2. AppInfo

Information about the remote application (→ `GetAppInfo`).

Fields:

- `id: str`
- `description: str`
- `version: str`
- `file_ext: str`
- `interactive: bool`

11.9.3. RecordingStatus

Status of the current recording (→ `GetRecordingStatus`).

Fields:

- `data_source: str`
- `duration_seconds: int`
- `file_size: int`

duration (property) → datetime.timedelta

`duration_seconds` as a `datetime.timedelta` (exact).

11.9.4. RunningTask

A background task running in the analyzer (→ `GetRunningTasks`).

Fields:

- `name: str`
- `progress_percent: int`
- `progress_applicable: bool`

progress (property) → int | None

Percent complete, or `None` when the server says progress isn't meaningful.

11.9.5. TraceFileInfo

Information about the loaded trace file (→ `GetTraceFileInfo`).

Fields:

- `path: str`
- `file_size: int`
- `start_datetime_utc_raw: int`
- `software_version: str`
- `data_source: str`
- `unique_id: str`
- `recording_options: str`

started_at_utc (property) → datetime.datetime | None

The trace start time as a tz-aware UTC datetime, or `None` when the raw value is non-positive (unset).

`start_datetime_utc_raw` is milliseconds since 2000-01-01 UTC (the epoch the analyzer's own trace indexer uses); reinterpret the raw value yourself if you ever need a different one.

11.10. Logic signals**11.10.1. LogicSignalTransitionType**

Which edge to search for (mirrors the `Slice LogicSignalTransitionType` enum).

Members: `ANY`, `RISING_EDGE`, `FALLING_EDGE`

11.10.2. LogicSignalTransition

A found logic-signal transition (→ `FindLogicSignalsTransition`).

Fields:

- `state: int`
- `time_ps: int`

time (property) → datetime.timedelta

The transition time as a `datetime.timedelta` (microsecond resolution).

11.11. Errors

11.11.1. RemoteControlError

Base class for every error raised by the SDK.

11.11.2. ConnectionFailed

The analyzer could not be reached or the connection was lost.

Common causes: the Ellisys software is not running, the Remote Control plugin is not enabled, or the host/port is wrong.

11.11.3. OperationError

The analyzer accepted the request but rejected the operation.

Wraps the Slice `OperationFailed` exception; the message carries the analyzer's reason when one is available.

11.11.4. NotAvailable

A requested product facet is not available on the connected analyzer.

Raised by `analyzer.bluetooth` (and future product facets) when the endpoint is reachable and healthy but is a **different** product, so the product interface's `checkedCast` returns no proxy. Distinct from `ConnectionFailed` (transport) and `OperationError` (the server rejected an operation, e.g. a wrong-product `export`); still catchable as `RemoteControlError`.

11.11.5. LoadTimeout

A trace load did not finish within the timeout.

Raised by `Analyzer.load` / `Analyzer.wait_until_loaded` when `isLoading` stays `True` past the deadline. Subclasses `OperationError` so existing `except OperationError` handlers still catch it; the message carries the filename (when known), the elapsed time, and the timeout.

Revisions History

Date	Rev	Changes
June 8, 2026	1.0	Initial Python SDK guide.
June 10, 2026	1.1	Added the Using with AI agents and Overview queries sections.

Copyright and Intellectual Property

Copyright Disclaimer

Copyright © Ellisys 2026. All Rights Reserved.

No part of this document or any of its contents may be reproduced, copied, modified or adapted, without the prior written consent of Ellisys.

Intellectual Property Disclaimer

This document is provided to you "as is" with no warranties whatsoever, including any warranty of merchantability, non-infringement, or fitness for any particular purpose. Ellisys disclaims all liability, including liability for infringement of any proprietary rights, relating to use or implementation of information in this document. The provision of this document to you does not provide you with any license, express or implied, by estoppel or otherwise, to any intellectual property rights.

Open Source Disclaimer

This Ellisys analysis software automation API plugin is based on a third-party networking library from ZeroC named ICE (<https://zeroc.com/products/ice>). This library is licensed under the GNU GPL v2 or later. This plugin is also licensed under the GPL, and its source code can be requested at gpl@ellisys.com.

This plugin is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.