VisualEther

User Manual

Version 9.0

June 25, 2026

EventHelix.com Inc.

Table of Contents

1 Introduction

1.1 What Is VisualEther?

VisualEther — Wireshark PCAP Analysis for Humans and AI.

VisualEther converts Wireshark packet captures (PCAP/PCAPNG files) into sequence diagrams. It extracts protocol messages, tracks session lifecycles, and renders the result as PDF for review or as HTML, NDJSON, and Markdown for richer workflows.

Whether you are debugging a BGP peering issue, analyzing 5G signaling flows, or validating SIP call setup, VisualEther turns raw packet data into clear diagrams that make interactions between network entities easier to understand.

The Professional and Server editions include a built-in MCP server, enabling Claude Code — the supported AI coding agent — to analyze protocol interactions directly. Rather than feeding raw packet dumps to the agent, VisualEther extracts only the fields that matter, so Claude Code can debug protocol issues, compare sessions, and verify feature behavior through natural-language conversation.

1.2 Key Concepts

1.2.1 Packet Captures (PCAP)

A PCAP file is a standard format for recording network traffic. VisualEther uses Wireshark's command-line tool `tshark` to dissect packets, so any protocol that Wireshark can decode is supported.

1.2.2 Field Extraction Templates (FXT)

FXT (Field Extraction Template) is an XML configuration format that tells VisualEther:

- **What messages to extract** from a packet capture
- **How to display them** — labels, parameters, and styling
- **How to group them** into sessions for lifecycle tracking
- **How to classify outcomes** — success, failure, incomplete

There are two types of FXT files, conventionally named `explore.fxt.xml` and `sessions.fxt.xml`:

Type	Purpose	Session Tracking
Explore	Visualize message flow in a single diagram	No
Sessions	Track request/response lifecycles per session	Yes

1.2.3 Sessions

A session represents a sequence of related messages between network entities. Examples include TCP connections, DNS query/response pairs, BGP peering sessions, SIP dialogs, and 5G NGAP signaling

flows. VisualEther groups messages into sessions, tracks their lifecycle from start to end, and classifies each session's outcome.

1.2.4 Sequence Diagrams

The primary output is a sequence diagram showing messages exchanged between network entities over time. Each arrow represents a protocol message, annotated with the message type and extracted parameters. An explore FXT produces a single sequence diagram covering all traffic, while a sessions FXT generates a separate sequence diagram for each tracked session.

1.3 How It Works

There are two ways to drive VisualEther:

- **Manual CLI** (every edition) — pick a built-in sample with `visualether list`, bootstrap a project with `visualether new`, refine the FXT and `hosts.txt` files against your own capture, then run `visualether generate`.
- **With Claude Code** (Professional / Server) — point Claude Code at a PCAP and it generates the FXT and `hosts.txt`, runs VisualEther, and reads the resulting diagrams and NDJSON to answer your questions in natural language.

Section 2 walks through both paths end to end. The rest of the manual covers the FXT format, the output formats, troubleshooting, and a curated collection of real-world templates.

1.4 Editions

VisualEther is available in three editions, which the running binary identifies in its startup banner:

- 🌱 **Community Edition** — runs whenever no license is installed.
- 💼 **Professional Edition** — a paid Professional license or an active Professional trial license unlocks all Professional features described in this manual.
- 🖥️ **Server Edition** — a paid Server license unlocks the same capabilities as Professional and additionally grants the right to install on one shared server, VM, or CI build agent, while bundling three developer seats. In one line: Professional is for one developer on up to two personal machines, while Server covers a small team plus one shared automation host.

The Community Edition is designed to teach protocol flows using sequence diagrams. Professional / Server editions add CI/CD workflows and AI-based analysis. The differences between the free and paid tiers are summarized below:

Feature	🌱 Community	💼 Professional	🖥️ Server
Output formats	PDF only	PDF, HTML, NDJSON, Markdown	PDF, HTML, NDJSON, Markdown
Paper sizes	A4, Letter	All sizes + custom dimensions	All sizes + custom dimensions
Page limit	10 pages	Unlimited	Unlimited

Diagram axes (entities)	10 max	Up to 64 per PDF (tunable)	Up to 64 per PDF (tunable)
Session tracking	—	✓	✓
MCP server (AI integration)	—	✓	✓
Cross-file session continuity	—	✓	✓
HTTP serving / auto-browser	—	✓	✓
Interactive workstation use	PDF viewing only	1 developer, up to 2 machines	3 developers, up to 2 machines each
Shared CI / server install	—	Not permitted	1 shared server, VM, or CI build agent
Best fit	Learning and small captures	Individual engineer / consultant	Small team + shared automation
FXT templates	All 82+ (subject to above limits)	All 82+	All 82+
PDF watermark	Yes	None	None

1.5 Community Edition Quick Start

If you are running without a license, start from a built-in sample so you get a working FXT and a known-good capture in one step:

```
# 1. See what protocols ship with built-in templates
visualether list

# 2. Bootstrap a project from the BGP sample
visualether new bgp
cd bgp

# 3. Generate the explore PDF
visualether generate
```

Inside a project directory created by `visualether new`, `visualether generate` auto-discovers packet captures and runs the available FXTs. In Community Edition, `sessions.fxt.xml` is skipped, so the main result is the explore PDF. Open the generated PDF directly in your browser or PDF viewer — no HTTP server is required. To use your own capture, replace the sample PCAP with your file, edit `explore.fxt.xml` to match the protocol fields you want to extract, and re-run `visualether generate`.

When later chapters discuss session tracking, MCP / Claude Code workflows, or HTML / NDJSON / Markdown output, treat them as Professional / Server only unless stated otherwise — the feature table above is authoritative on what’s excluded.

2 Getting Started

> **Tip**
>
> Running without a license? See Section 1.5 for the Community Edition workflow. The Claude Code walkthrough below requires a Professional or Server license; the Manual CLI walkthrough works on every edition (sessions are skipped on Community).

2.1 Installing VisualEther

VisualEther installs through the native package manager on each platform, so the commands always fetch the current release. On Linux you register the EventHelix package repository once — after that, the initial install and every future upgrade go through the package manager normally, with no need to repeat the setup step.

> **Important**
>
> Each command in this section is a single line. Some are long enough that they wrap across more than one printed line to fit the page width — those breaks are an artifact of the manual's form factor, not part of the command. When you copy a command, enter it as one continuous line and remove any line break introduced by the wrapping.

2.1.1 Windows

Install with winget, the Windows Package Manager built into Windows 10 and 11:

```
winget install EventHelix.VisualEther
```

This places `visualether` on your `PATH` so the commands throughout this manual work from any terminal (PowerShell, Windows Terminal, or Command Prompt). To upgrade later:

```
winget upgrade EventHelix.VisualEther
```

Alternatively, download the signed Windows installer from the VisualEther download page and run it; to update, download the latest installer and run it over your existing copy.

VisualEther needs Wireshark's `tshark` to read captures; install Wireshark for Windows from wireshark.org (see Section 2.2). Supported on Windows 10 and later (`x86_64`).

2.1.2 macOS (Apple Silicon)

Install from the EventHelix Homebrew tap:

```
brew install eventhelix/tap/visualether
```

Homebrew also installs Wireshark's `tshark`, which VisualEther uses to read captures. The macOS build supports Apple Silicon (`arm64`) only.

To upgrade later:

```
brew upgrade visualether
```

2.1.3 Linux — Ubuntu and Debian (apt)

First, register the EventHelix apt repository. This is a one-time step that imports the signing key and adds the package source:

Import the signing key:

```
curl -fsSL https://downloads.eventhelix.com/apt/eventhelix.gpg | sudo gpg --
dearmor -o /usr/share/keyrings/eventhelix.gpg
```

Then add the package source[1]:

```
echo "deb [signed-by=/usr/share/keyrings/eventhelix.gpg] https://downloads.
eventhelix.com/apt stable main" | sudo tee /etc/apt/sources.list.d/
eventhelix.list
```

Then install VisualEther:

```
sudo apt update && sudo apt install visualether
```

Supported on Ubuntu 22.04 LTS and later and Debian 12 and later (`x86_64`). The package recommends `tshark`, which a standard apt configuration installs automatically.

To upgrade later, do **not** repeat the repository setup — it stays registered. A normal update installs the current release:

```
sudo apt update && sudo apt upgrade visualether
```

2.1.4 Linux — Fedora (dnf)

First, register the EventHelix dnf repository. This is a one-time step that adds the package source and its signing key:

```
sudo curl -fsSL https://downloads.eventhelix.com/rpm/eventhelix.repo -o /etc/
yum.repos.d/eventhelix.repo
```

[1] `tee` takes one argument here — `/etc/apt/sources.list.d/eventhelix.list`.

Then install VisualEther:

```
sudo dnf install visualether
```

Supported on the current and previous Fedora releases (`x86_64`). The package installs `wireshark-cli` — which provides `tshark` — automatically.

To upgrade later, do **not** repeat the repository setup — it stays registered:

```
sudo dnf upgrade visualether
```

2.1.5 Manual installation (direct downloads)

Signed installers and archives for every platform — the Windows installer, the macOS package and disk image, and `.tar.gz` , `.deb` , and `.rpm` archives — are available from the VisualEther download page for manual download.

> **Tip**
>
> Installed more than once? If `visualether --version` reports an older release after an install or upgrade, you most likely have several `visualether` binaries on your `PATH` and the shell is running a stale one. List them with `which -a visualether` (macOS/Linux) or `where visualether` (Windows), then remove the old copy — see Section 16.13.

2.2 Prerequisites

2.2.1 Wireshark / tshark

VisualEther requires `tshark` (Wireshark's command-line dissector) to parse packet captures. Install Wireshark from wireshark.org. The `tshark` binary is included with the standard installation.

We recommend **Wireshark 4.6 or later**. VisualEther surfaces whatever fields `tshark` reports, and newer Wireshark releases ship richer dissectors. Some templates rely on fields that older releases do not emit — for example, the HTTP/3 templates key on a per-stream field that is missing in older Wireshark such as 4.2 — and when a field is absent the affected messages or sessions drop out of the diagram silently rather than reporting an error. VisualEther flags this for you: `visualether generate` and `visualether analyze` print a warning when they detect a `tshark` older than the recommended version.

Verify that `tshark` is installed and check its version:

```
tshark --version
```

On Linux, the Wireshark version in your distribution's package repository may be older than recommended — some long-term-support releases still ship Wireshark 4.2. If `tshark --version` reports an older release, install a current Wireshark following the instructions at wireshark.org.

If `tshark` is not on your `PATH`, you can specify its location via the `VISUALETHER_TSHARK_PATH` environment variable (see Section 18.14).

2.3 Your First Diagram with Claude Code

The fastest way to get started is to let Claude Code handle the workflow. At the time of writing, the VisualEther MCP server has been tested with Claude Code using Claude Opus 4.7. This walkthrough uses an MP-BGP capture from weberblog.net. Download the capture and place the unzipped file `BGP-dump FINAL.pcap` in a new `mp-bgp` directory.

> **Tip**
>
> BGP decodes out of the box because `tshark` recognizes it on standard ports. Some protocols (notably radio-layer captures using `DLT_USER` framing, and certain non-standard ports) need extra dissector hints — the Section 4 sample preloads them via `visualether.toml`, and Section 16.4 covers the troubleshooting path. One of those preloaded hints, `nas-5gs.null_decipher:TRUE`, is safe only for null-ciphered (NEA0) test captures; if you reuse that sample's `visualether.toml` on a real-ciphered capture, remove it (see Section 4).

Install the VisualEther MCP server for the project (see Section 14 for details):

```
visualether mcp install --scope project
```

Launch Claude Code from the `mp-bgp` directory. The code blocks below show example prompts to enter in Claude Code.

First, ask Claude Code to create FXT templates from the PCAP file:

```
Please create explore.fxt.xml and sessions.fxt.xml from @"BGP-dump FINAL.pcap"
```

Claude Code analyzes the capture, detects MP-BGP traffic over both IPv4 and IPv6, and generates two FXT templates — an `explore.fxt.xml` for visualizing all BGP messages in a single diagram and a `sessions.fxt.xml` for tracking individual BGP peering sessions with lifecycle outcomes. As the closing step Claude Code also writes a `visualether.toml` and a matching `.gitignore` into the same directory, so the project is fully configured: `visualether generate` works from any directory without flags. (For radio captures, the toml also carries the right `tshark-args` for the DLT_USER linktype — see Section 16.4.)

Next, ask Claude Code to generate the diagrams:

```
Generate the explore diagram
```

```
Generate the sessions diagram.
```

Both diagrams open automatically in your browser. The explore diagram (not shown here) contains all BGP sessions from the PCAP file in a single sequence diagram. The sessions diagram produces two

interactive views: a **Session Navigator** that indexes every tracked session, and a **session viewer** that shows the sequence diagram plus a detail panel for a single session. Both are shown below (Figure 1 and Figure 2; full reference in Section 12.8 and Section 12.2). Click any session timeline bar in the Session Navigator to open that session's sequence diagram in the session viewer.

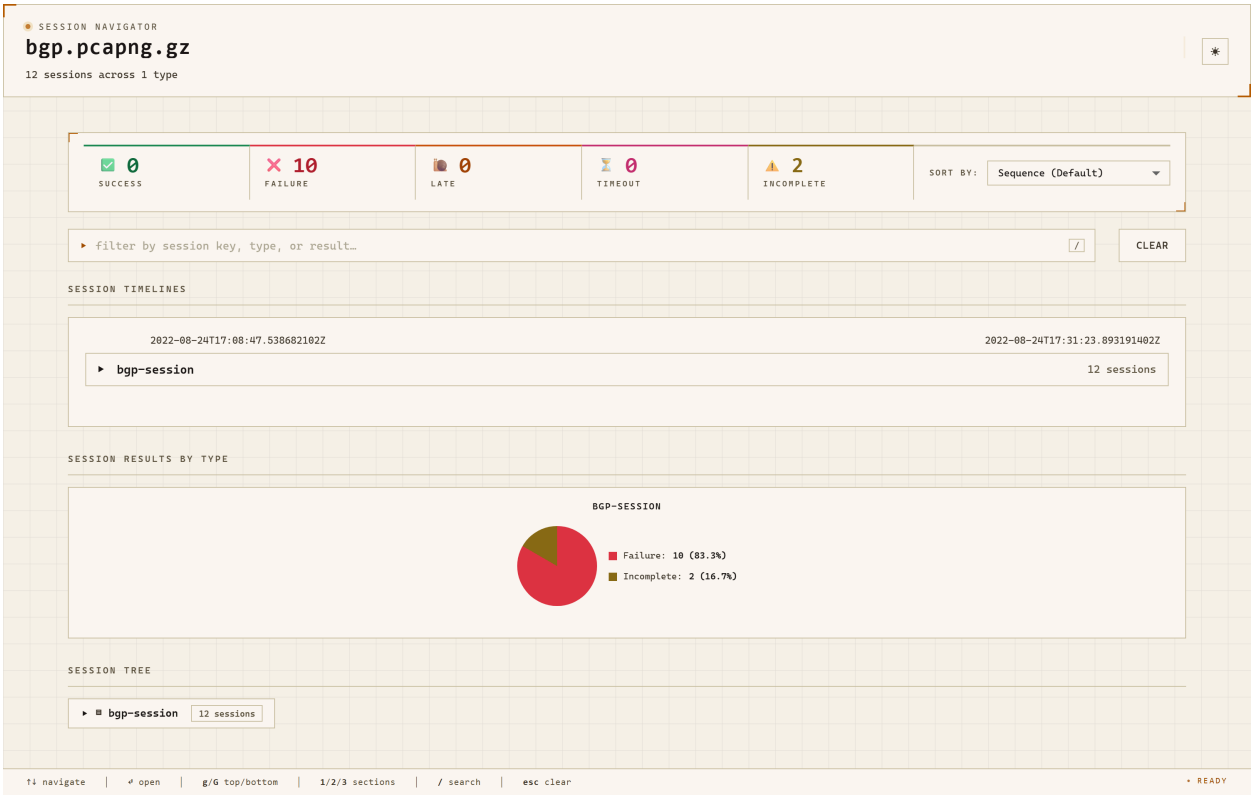

Figure 1: MP-BGP Session Navigator — click a session timeline bar to open the session viewer.

Clicking a message in the session viewer scrolls the detail panel on the right to that message, where an expandable tree shows every field in the packet.

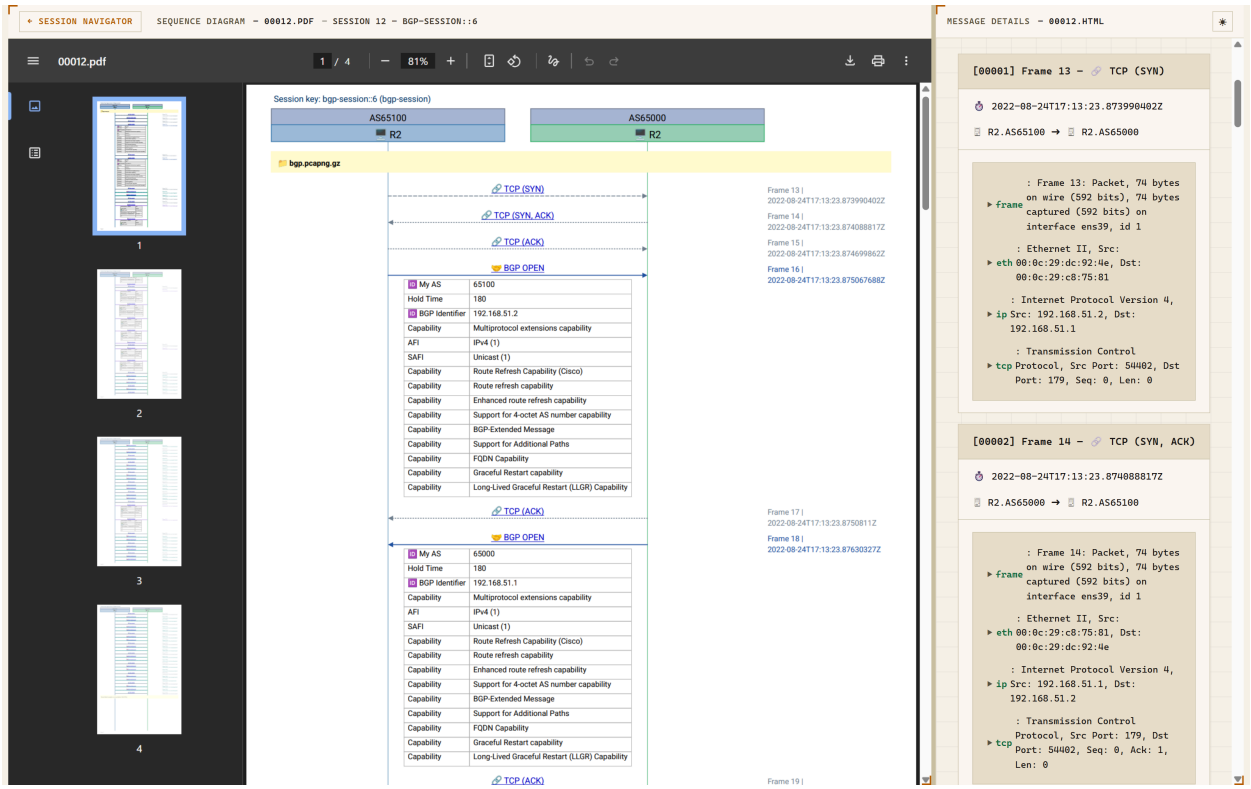

Figure 2: MP-BGP Session Viewer — click a message in the sequence diagram to see its details.

Claude Code can also generate a hosts.txt file that maps raw IP addresses to router names, derived from BGP Identifier fields in OPEN messages, making the diagrams easier to read. Ask Claude Code:

```
Please generate the hosts.txt file.
```

```
Regenerate the explore and sessions diagrams with the hosts.txt file.
```

Once the setup files are in place, day-to-day runs usually regenerate NDJSON and diagrams from new captures. Claude Code can then read the NDJSON output files and answer follow-up questions directly, so there is no need to inspect diagrams or write scripts manually. For example:

```
What is the average time between a TCP connection establishment and the first
BGP Open message?
```

```
Why did BGP sessions fail?
```

> **Tip**
>
> This is the recommended workflow for most users. The manual CLI workflow below is useful when you need fine-grained control over templates.

2.4 Your First Diagram (Manual CLI)

This walkthrough sets up the same FXT templates directly from the CLI, starting from a built-in sample.

2.4.1 Step 1 — List available samples

VisualEther ships with built-in protocol templates for many protocols. List them with:

```
visualether list
```

2.4.2 Step 2 — Create a project from a sample

Create a new project directory from the BGP sample:

```
visualether new bgp
cd bgp
```

This creates a working BGP project containing:

- An `explore.fxt.xml` (single-diagram template)
- A `sessions.fxt.xml` (session-tracking template)
- A `hosts.txt` file mapping the sample's router IPs to names like `R1.AS65000` and `R2.AS65100`
- A `visualether.toml` project configuration. `generate` also auto-discovers `hosts.txt` in the project directory, so diagrams render with friendly entity names automatically
- The built-in sample PCAP `bgp.pcapng.gz`

You can generate diagrams immediately from the sample data, and the FXT and `hosts.txt` files are then ready to use as a starting point for your own BGP PCAP files.

Inside a project directory created by `visualether new` (or `visualether init`), `visualether generate` with no arguments auto-iterates over `explore.fxt.xml` / `sessions.fxt.xml` and auto-discovers any `.pcap` / `.pcapng` / `.pcap.gz` / `.pcapng.gz` files. In Community Edition, `sessions.fxt.xml` is skipped. See Section 18 for the full flag list.

2.4.3 Step 3 — Generate diagrams from your PCAP

Copy your PCAP file into the directory (this example uses the same `BGP-dump FINAL.pcap` from the Claude Code section above). Both commands can use the same `--output` directory because VisualEther writes results under a subdirectory named after the input capture stem.

The short form relies on the `visualether.toml` written by `new`: `output` comes from the toml; the FXT files (`explore.fxt.xml` and `sessions.fxt.xml`) and PCAP files in the project dir are picked up by `generate` 's auto-iteration and auto-discovery. One command renders both diagrams:

```
visualether generate --serve
```

To regenerate from only one FXT, pin it on the command line:

```
visualether generate --fxt sessions.fxt.xml --serve
```

For scripting against captures outside the project directory, pass `--fxt`, `--input`, and `--output` explicitly — see Section 18 for the full form.

The `--serve` flag (Professional / Server) auto-spawns `visualether serve` for the output directory and opens the **Capture Atlas** (Figure 3) — a tree-view landing page that links to every generated diagram — in your default browser. The explore diagram produces a single sequence diagram containing all BGP traffic. The sessions diagram produces per-session sequence diagrams accessible from the Session Navigator. On Community Edition, omit `--serve` and open the generated PDF directly.

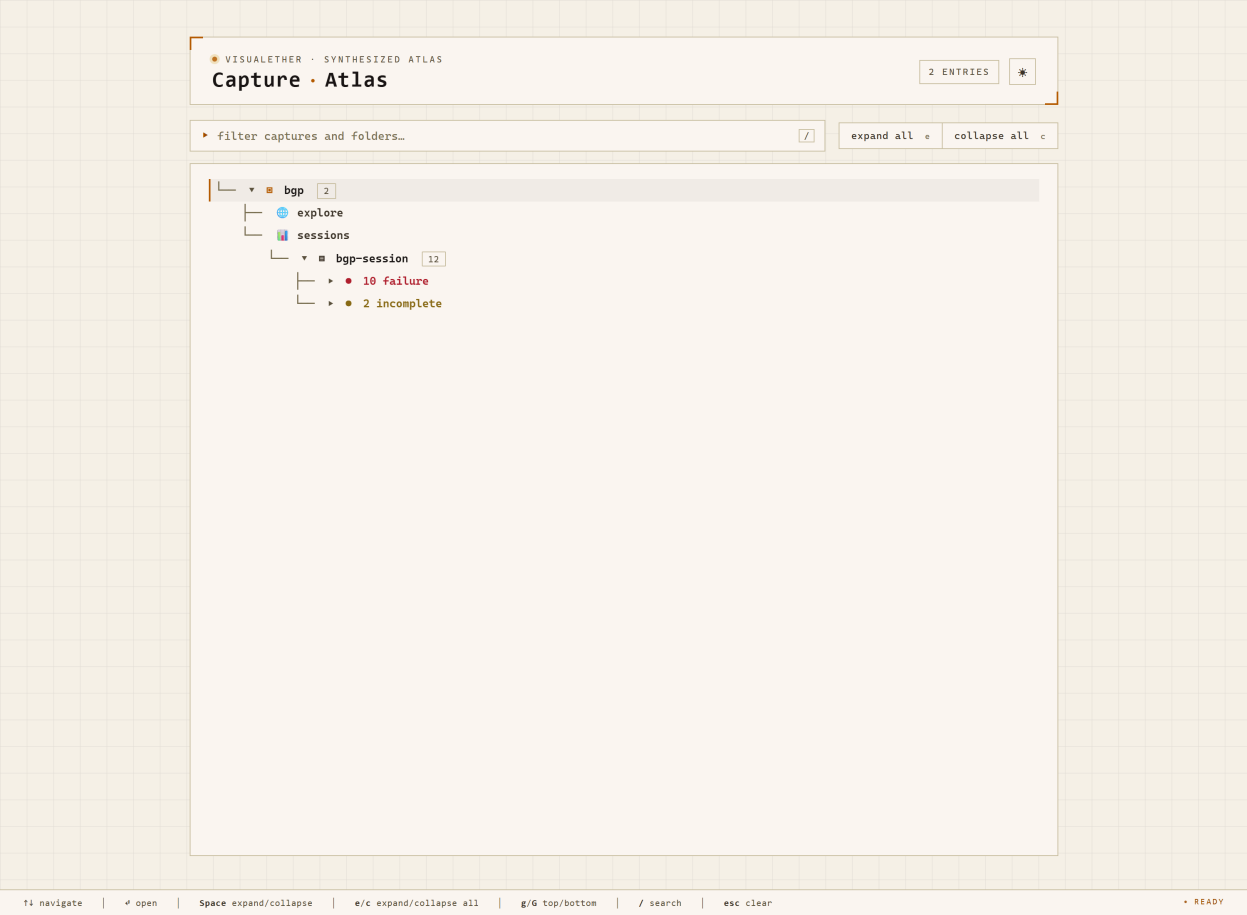

Figure 3: Capture Atlas — the landing page served at `/`, linking to every generated diagram under the output directory.

2.4.4 Step 4 — Explore the output

With both the sample `bgp.pcapng.gz` and your `BGP-dump FINAL.pcap` in the project directory, the output layout looks like this:

```
output/
  bgp/              # From bgp.pcapng.gz (sample, IPv4 only)
    bgp_viewer.html  # Combined HTML+PDF viewer
    bgp.pdf          # Standalone PDF diagram
    bgp.html         # HTML message details
```

```
    bgp-session/     # Per-session diagrams
    index.html       # Session Navigator
  BGP-dump FINAL/  # From BGP-dump FINAL.pcap (MP-BGP, IPv4 + IPv6)
    BGP-dump FINAL_viewer.html
    BGP-dump FINAL.pdf
    BGP-dump FINAL.html
    bgp-session/     # IPv4 BGP sessions
    bgp-session-v6/  # IPv6 BGP sessions
    index.html       # Session Navigator
```

Each capture lands in its own per-stem subdirectory; explore output and per-session output share the same subdirectory. The Capture Atlas you reached via `--serve` is synthesized at `/` by `visualether serve` — it is not written to disk.

2.4.5 Step 5 — Customize templates with the Field Navigator

To refine the generated FXT templates for your specific capture, use `visualether analyze` to discover protocol fields:

```
visualether analyze "BGP-dump FINAL.pcap"
```

This opens the Field Navigator in your browser (Figure 4) — a hierarchical tree of every protocol field in the PCAP, with example values and ready-to-paste `<opcode>` / `<param>` snippets. See Section 12.9 for the annotated walkthrough.

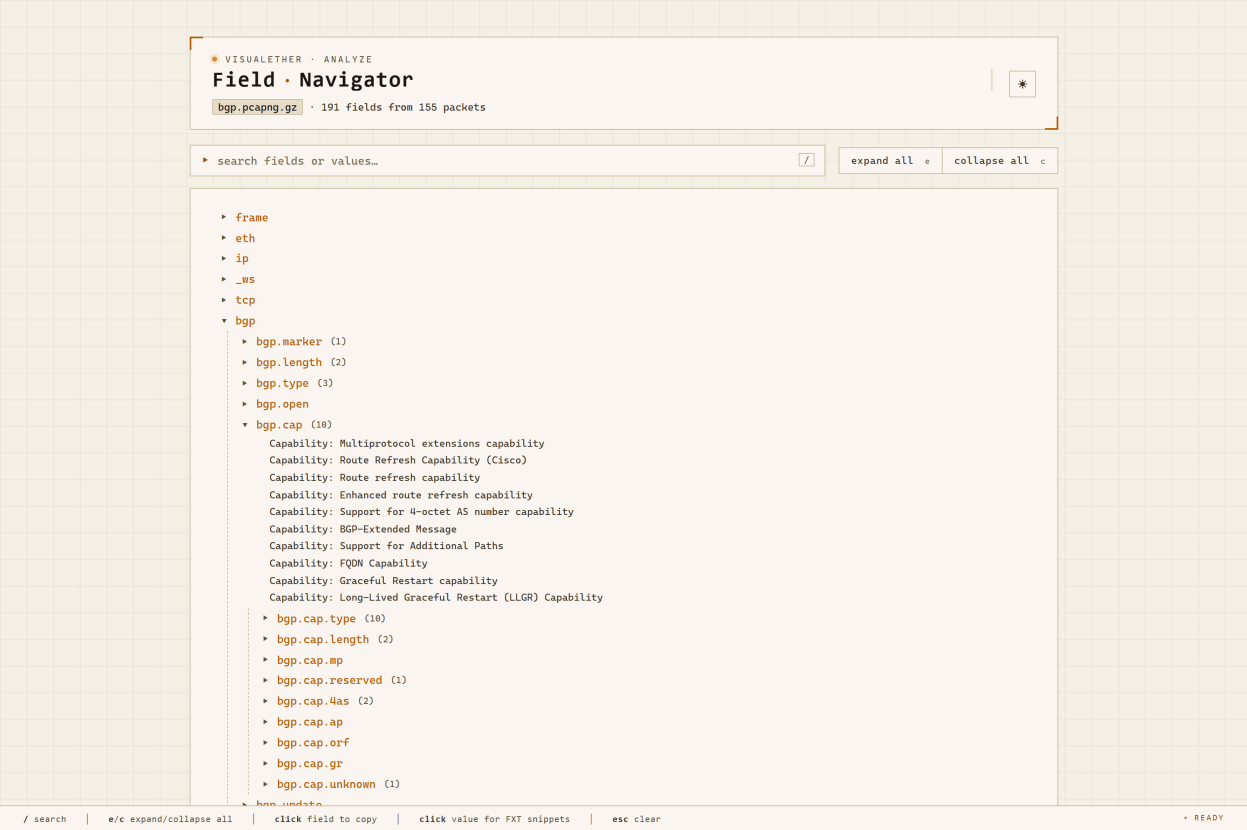

Figure 4: MP-BGP Field Navigator — click a field value to copy opcode or param snippets for your FXT template.

Edit the `explore.fxt.xml` and `sessions.fxt.xml` files that `visualether new` copied from the BGP sample to add, remove, or modify opcode and param elements based on what you see in the Field Navigator. See Section 12.9 for an annotated overview and Section 3 for a step-by-step guide to FXT template syntax.

> **Tip**
>
> `sessions.fxt.xml` and `hosts.txt` are prepared during setup, not rewritten for every run. Once they are in good shape, your CI/CD pipeline can repeatedly call `visualether generate` to produce fresh NDJSON and diagrams from new captures.

2.5 Next Steps

After you have generated your first diagrams, use the following chapters to deepen your workflow:

- **Learn the FXT basics** — Section 3 walks through explore and sessions FXT templates step by step, and Section 5 explains the FXT format in depth.
- **Use AI-driven workflows** — Section 14 covers MCP server setup and configuration, and Section 15 shows practical examples of protocol debugging with Claude Code.

- **Look up commands and protocol patterns** — Section 18 covers all command-line options and the `visualether.toml` configuration file, and Section 13 shows how to bootstrap an FXT for a protocol you have not seen before.
- **Study real-world examples** — Section 17 points to the Nick Russo PCAP Diagrams collection for hands-on learning in both Community and Professional editions.

3 Tutorial

VisualEther ships with a self-guided tutorial that introduces FXT concepts through a real TCP/HTTP packet capture. The tutorial files contain extensive `TUTORIAL:` inline comments that explain each element at the point of use — open them in any text editor and read them alongside this chapter.

> **Tip**
>
> Add `--serve` to any `generate` command below to auto-spawn `visualether serve` and open the **Capture Atlas** (Figure 3) in your default browser (Professional / Server). In Community Edition, omit `--serve` and open the generated PDF in `output/`.

To set up a local working directory with the tutorial files:

```
visualether new tutorial
cd tutorial
```

This creates a `tutorial/` directory containing the sample PCAP, FXT templates, and hosts file. The tutorial is embedded as comments inside the FXT and hosts files — open them in any text editor to follow along. It is split across three files:

File	Level	What You Learn
`explore.fxt.xml`	Beginner	Message matching, opcode, params, styling, template ordering
`sessions.fxt.xml`	Intermediate	Session lifecycle, multi-match, qualify, keys, outcome classification
`hosts.txt`	Reference	IP-to-name mapping, hierarchy, axis coalescing, IPv6 port notation

3.1 Sample Capture

The tutorial uses **tcp-sack-video.pcapng**, an HTTP video download from a CDN server that exhibits TCP SACK (Selective Acknowledgment) behavior — including duplicate ACKs, retransmissions, out-of-order segments, and window-full conditions. This mix of TCP events makes it an ideal learning capture.

Property	Value
Packets	202
Duration	0.587 seconds
Client	10.0.0.145 (Firefox browser)

Server	186.15.230.24 (Apache on port 80)
Protocols	HTTP/1.1, TCP

3.2 Part 1: Explore FXT — Visualizing Packet Flow

An **explore FXT** produces a single sequence diagram showing every matched packet in chronological order. There is no session tracking — it simply visualizes message flow between endpoints.

Open `explore.fxt.xml` in any text editor and follow the inline comments as you read through this section.

Generate the explore diagram with this command:

```
visualether generate \
  --fxt   explore.fxt.xml \
  --input tcp-sack-video.pcapng \
  --output output
```

3.2.1 The FXT Root Element

Every FXT file starts with a single `<FXT>` root element. In default single-match mode (no root attributes), every packet matches at most one template, selected by priority and then file order. The only optional root attribute relevant to selection is `multi-match`:

Attribute	Description
`multi-match="true"`	Allow a single packet to match multiple templates (demo in Part 2; formal definition in Section 5.2)

When multiple templates could match the same packet — e.g., both an HTTP template and a generic TCP template match an HTTP GET — the template listed earlier in the file wins (first-match-wins by file order). The tutorial's `explore.fxt.xml` places HTTP templates before the TCP catch-all so HTTP GET packets get an HTTP label and everything else falls through to TCP. A `priority="N"` attribute on individual templates overrides file order when needed (see Section 5.3).

3.2.2 Message Elements

VisualEther provides one message element per transport protocol. The tutorial capture is HTTP over TCP/IPv4, so every template uses `<tcp-message>`:

Element	Transport	Example Protocols
`<tcp-message>`	TCP/IPv4	HTTP, BGP, TLS, SSH

The transport element determines how source and destination addresses are extracted for the sequence diagram axes. For example, `<tcp-message>` uses `ip.src` / `ip.dst` and `tcp.srcport` /

`tcp.dstport`. See Section 6 for the full list of message elements (UDP, SCTP, IPv6 variants, MAC, WiFi, generic).

3.2.3 Opcode and Regex Transformations

The `<opcode>` element is the heart of every message template. It serves two roles:

1. **Matching** — The text content is a Wireshark field name. If a packet contains that field, the template becomes a candidate match.
2. **Labeling** — The field's value becomes the message label in the sequence diagram.

> **Tip**
>
> **How to find Wireshark field names:**
> - `visualether analyze <pcap>` — opens the Field Navigator, where you can copy field codes or full `<opcode>` and `<param>` snippets
> - In Wireshark: right-click a field → Copy → Field Name
> - In the combined viewer, click a message arrow to see the field tree on the right side — field codes can be copied from there

Raw Wireshark values are often verbose, so `match` and `replace` help transform them:

```
<opcode
    match="(.*)"
    replace="HTTP $1"
>http.request.method</opcode>
```

Attribute	Description
`match`	Regex pattern tested against the field value. Capture groups `( )` extract parts.
`replace`	Builds the display label. `$1` = first capture group, `$2` = second, etc.

Examples from the tutorial:

Field Value	match	replace	Result
`Request Method: GET`	`(.*)`	`HTTP $1`	`HTTP Request Method: GET`
`Flags: 0x002 (SYN)`	`.*\(SYN\)$`	`TCP SYN`	`TCP SYN`
`Flags: 0x012 (SYN, ACK)`	`.*\(SYN, ACK\)$`	`TCP SYN-ACK`	`TCP SYN-ACK`
`Flags: 0x018 (PSH, ACK)`	`.*\((.*)\)`	`TCP $1`	`TCP PSH, ACK`

The patterns above are simplified for clarity. The shipped tutorial files use a richer normalization regex (`^(?:.*?= )?(?:[^:]*:\s*)?(.*?)(?:\s*\[bit length[^\]]*\])?$`) that strips Wireshark prefixes and bit-length suffixes, and the `replace` strings carry emoji prefixes by convention — e.g. `📤 HTTP $1` , `📏 Length: $1` , `🔢 Seq: $1` , `✅ Ack: $1` , `🔁 Dup-ACK#: $1` . The modifier emoji is always on the left of the noun.

> **Tip**
>
> Use `display="brief"` on the opcode to get shorter field values before the regex is applied. For example, `"GET"` instead of `"Request Method: GET"` , producing `"HTTP GET"` after the replacement.

> **Tip**
>
> The `analyze` command's Field Navigator can auto-generate `match` and `replace` attributes for common transformation scenarios.

3.2.4 Parameters

Each `<param>` extracts an additional Wireshark field, which is displayed as a parameter line beneath the message arrow:

```
<tcp-message style="royal-blue">
  <opcode>http.request.method</opcode>
  <param>http.request.uri</param>
  <param>http.host</param>
  <param>http.user_agent</param>
</tcp-message>
```

- Any number of params per template
- If the field is absent from a packet, the param is silently skipped
- Params also support `match` / `replace` for value transformation

3.2.5 Styling

The `style` attribute controls the arrow color and line pattern:

```
style="royal-blue"              <!-- Solid blue -->
style="forest-green dashed"     <!-- Dashed green -->
style="brick wave"              <!-- Wavy red (errors) -->
style="slate dotted"            <!-- Dotted gray (background) -->
style="rgb(255,128,0) dashed"   <!-- Custom orange, dashed -->
```

See Section 10 for the full 30-color palette and the complete arrow-style list (`dashed` , `dotted` , `dot-dash` , `wave` , plus `regular` for an explicit solid arrow). The tutorial uses the following convention:

- **Regular** (no modifier) = requests and events
- **Dashed** = responses
- **Wave** = errors (RST, lost segments)
- **Dotted** = background or transport traffic

3.2.6 Bookmarks

`bookmark="true"` creates a PDF bookmark for the message, making navigation in the PDF viewer easier:

```
<tcp-message style="crimson" bookmark="true">
  <opcode match=".*" replace="TCP Fast Retransmission">
    tcp.analysis.fast_retransmission
  </opcode>
</tcp-message>
```

Bookmark key events such as SYN, FIN, RST, HTTP requests, and retransmissions, and skip high-volume messages such as generic ACKs.

3.2.7 Remarks — Inline Annotations

`<remark>` attaches an annotation to a message — used in the tutorial to explain **why** each TCP anomaly happens, not just **that** it happened. Each retransmission-family template (Duplicate ACK, Fast Retransmit, RTO Retransmission, Spurious Retransmission, Out-of-Order, Lost Segment, Window Full) carries a 💡 lightbulb remark with the underlying mechanism, the relevant RFC, and the typical root causes:

```
<tcp-message style="crimson" bookmark="true">
  <opcode match=".*" replace="⚡ TCP Fast Retransmission">
    tcp.analysis.fast_retransmission
  </opcode>
  <param match="(.*)" replace="🚀 Bytes-in-flight: $1">
    tcp.analysis.bytes_in_flight
  </param>
  <remark
    match=".*Frame Number: (\d+)"
    replace="💡 [Frame $1] Fast Retransmit (RFC 5681) --- after 3 duplicate
ACKs the sender resends the lost segment immediately rather than waiting for
the RTO timer..."
  >frame.number</remark>
</tcp-message>
```

Remarks render as a separate annotation block on the arrow, so the message label stays terse while the explanation lives alongside. The tutorial opens every lightbulb remark with 💡, making them easy to scan visually in the diagram.

3.2.8 Filter — Hiding Noise

`filter="true"` matches a packet (so it is still parsed and counted) but suppresses its arrow from the diagram. The tutorial uses this to drop pure ACK packets:

```
<tcp-message filter="true">
  <opcode match=".*\(ACK\)$" replace="🔔 TCP ACK">tcp.flags</opcode>
</tcp-message>
```

In a bulk transfer, pure ACKs (Flags=0x010, no payload) typically outnumber data segments and dominate the diagram without adding information. Filtering them keeps the focus on data segments and the SACK / retransmission events that explain the flow. Delete the filter template to restore them.

> **Warning**
>
> **Multi-match gotcha (sessions FXT).** With `multi-match="true"` (Part 2), every matching template generates a message — so a `filter="true"` template alone does **not** prevent the catch-all from also matching pure ACKs. The sessions template tightens the catch-all's regex to `.*\((.+,.+)\)` (requires a comma) so single-flag `(ACK)` no longer matches it.

3.2.9 Template Ordering — First-Match-Wins

Templates are checked from top to bottom. The first template whose opcode field exists in the packet wins.

Rules:
1. Place **specific** templates before **generic** catch-all templates
2. Place **higher-layer** templates (HTTP) before **lower-layer** ones (TCP)
3. The catch-all template (`tcp.flags` with `.*\((.*)\)`) must be **last**
4. Use `priority="N"` on a template to override file order when needed

The tutorial orders its templates as follows:

Position	Templates	Why
1st	HTTP (`http.request.method` , `http.response.code`)	Higher layer — should win over TCP for HTTP packets
2nd	TCP analysis (`tcp.analysis.duplicate_ack` , etc.)	Specific anomaly fields
3rd	TCP flags (SYN, SYN-ACK, FIN, RST)	Specific regex patterns on `tcp.flags`
4th	Pure-ACK filter (`filter="true"`)	Hides bare ACKs so the diagram stays focused on data and anomalies
Last	Generic TCP catch-all	Matches any remaining `tcp.flags`

3.3 Part 2: Sessions FXT — Lifecycle Tracking

A **sessions FXT** groups related messages into sessions with lifecycle management and outcome classification. Sessions are written to output directories organized by session type and outcome.

> **Warning**
>
> Session tracking is a Professional / Server feature. On Community Edition, `sessions.fxt.xml` is skipped, and Part 2 produces no output — read along to understand the FXT shape, then return when you have a Professional or Server license to run it.

Open `sessions.fxt.xml` in any text editor and follow the inline comments as you read this section.

Generate the session diagrams with the following command:

```
visualether generate \
  --fxt   sessions.fxt.xml \
  --input tcp-sack-video.pcapng \
  --output output
```

3.3.1 Multi-Match Mode

```
<FXT multi-match="true">
```

In Part 1 (default single-match mode), each packet matched exactly one template — the first one in file order whose opcode field was present (or the highest-priority match when `priority="N"` was set). **Multi-match** lets a single packet match **multiple** templates, with each match going to its own session type. (See Section 5.2 for the full semantics; this section walks the tutorial's use of the attribute.)

For example, an HTTP GET packet matches both the HTTP request template (creating an `http-session`) and the TCP catch-all template (adding a message to the `tcp-session`).

Mode	Behavior	Use Case
Default (single-match)	One match per packet, decided by file order + optional `priority="N"`	Explore diagrams
`multi-match`	Multiple matches per packet	Cross-layer session tracking

3.3.2 Session Filter

```
<session-filter outcome-in="success failure late timeout incomplete" />
```

Limits CLI output to specific outcome categories. Space-separated values:

Category	Description
`success`	Normally completed
`failure`	Error or abnormal termination
`late`	Response received after `max-duration` deadline
`timeout`	No response within `max-duration`
`incomplete`	Never properly closed

For illustration, the tutorial includes all outcome categories so you can see both the `tcp-session` (incomplete) and `http-session` (success) in the output. In production CI/CD pipelines, you would typically use `outcome-in="failure incomplete"` to focus on problems. Remove this element entirely to include all sessions (equivalent to listing every category).

3.3.3 Session Type Definition

A `<session-type>` defines how packets are grouped into sessions:

```
<session-type name="tcp-session" default="true">
  <qualify>...</qualify>
  <key>...</key>
  <!-- <fallback-key> is optional and rarely needed; see below -->
</session-type>
```

Attribute	Type	Default	Description
`name`	string	required	Unique identifier referenced by templates via `session-type="..."`
`default`	boolean	`false`	Templates without `session-type` use this session type (at most one)
`direction-agnostic`	boolean	`true`	Treat A→ B and B→ A as the same session. Set `="false"` only for directional protocols (e.g. ARP request→ reply, syslog client→ server)
`max-duration`	number	none	Timeout in seconds — sessions exceeding this are auto-closed
`auto-start`	boolean	`false`	Create session on first qualifying packet if none exists

3.3.4 Qualify — Packet Eligibility

The `<qualify>` element filters which packets are eligible for a session type. Without it, all packets matching the transport type would be eligible.

```
<qualify logic="or">
  <field value="80">tcp.srcport</field>
  <field value="80">tcp.dstport</field>
</qualify>
```

Logic	Description
`logic="or"`	Match if **any** condition is true
`logic="and"`	Match if **all** conditions are true (default)

Field constraint types:

Constraint	Description	Example
`value`	Exact match or comma-separated list	`value="80,443,8080"`
`value-min` / `value-max`	Inclusive numeric range (both required)	`value-min="1024" value-max="65535"`
`value-regex`	Regular expression pattern	`value-regex="^10\."`

3.3.5 Key — Session Identity

The `<key>` defines what makes a session unique. Three options:

Option A — Source/Destination pairs (used for the TCP session):

```
<key>
  <source>
    <field>ip.src</field>
    <field>tcp.srcport</field>
  </source>
  <destination>
    <field>ip.dst</field>
    <field>tcp.dstport</field>
  </destination>
</key>
```

Option B — Protocol identifier (an application-defined ID that scopes the session, e.g. `sip.Call-ID`, `diameter.Session-Id`, or `dns.id`):

```
<key>
  <protocol>
    <field>sip.Call-ID</field>
  </protocol>
</key>
```

For plain TCP / UDP / SCTP session types, the Wireshark stream index (`tcp.stream` , `udp.stream` , `sctp.assoc_index`) is the recommended default — a single field replaces the four-field source/destination pair and the bucketing matches Wireshark's own Conversations view. Under `--merge-inputs` VisualEther fronts dissection with mergecap, so these indices stay globally unique across the merged stream (see Section 8.3.6). Use the transport 4-tuple (`addr.src` +srcport / `addr.dst` +dstport) when you need participant axes derived directly from the key, or when composing with an app-layer ID.

Option C — Combined (source/destination plus protocol field):

```
<key>
  <source><field>ip.src</field></source>
  <destination><field>ip.dst</field></destination>
  <protocol><field>dns.id</field></protocol>
</key>
```

Rules:
- `<source>` and `<destination>` must appear together with equal field counts
- At least one of `<protocol>` or `<source>` / `<destination>` is required
- `direction-agnostic` is `true` by default, so (A→ B) and (B→ A) produce the same key. Set `="false"` for protocols where direction matters (ARP, syslog, etc.)

The `<fallback-key>` provides an alternative when primary key fields are genuinely absent for some packets (e.g., NAT or capture-location asymmetry):

```
<!-- Primary key is the 4-tuple; fall back to just IP pair if ports
     can't be dissected. -->
<fallback-key>
  <source><field>ip.src</field></source>
  <destination><field>ip.dst</field></destination>
</fallback-key>
```

For plain TCP/UDP/SCTP transports the primary 4-tuple is always present, so adding a fallback "just in case" is dead config — leave it out unless you have a concrete case where the primary fields are missing.

3.3.6 Session Lifecycle

Three attributes on message templates control the session lifecycle:

```
<!-- tcp-session is the default — no session-type needed -->
<tcp-message session-start="true">
  <opcode match=".*\((SYN)\)"
          replace="TCP ($1)">
    tcp.flags
  </opcode>
</tcp-message>

<!-- STOP: Closes the session with an outcome -->
<tcp-message session-stop="true"
             session-result="success::graceful-fin">
  <opcode match=".*\((.*FIN.*)\)"
          replace="TCP ($1)">
    tcp.flags
  </opcode>
</tcp-message>

<!-- Other session types still need explicit session-type -->
<tcp-message session-type="http-session"
             session-start="true">
  <opcode>http.request.method</opcode>
</tcp-message>
```

When `default="true"` is set on a session type, templates that omit `session-type` automatically belong to it. This reduces repetition when most templates belong to one session type — only templates for *other* session types need an explicit `session-type` attribute.

Lifecycle rules:
- A message **cannot** be both `start` and `stop`.
- Only `session-start="true"` messages create new sessions.
- Regular messages are added to **existing** sessions; discarded if none exists.
- Sessions never stopped get outcome `incomplete::no-close`.

Session result format: `category::reason`

Category	Example Reasons
`success`	`graceful-fin`, `2xx`, `3xx`, `completed`
`failure`	`tcp-reset`, `4xx`, `5xx`, `timeout`
`incomplete`	`no-close`, `mid-capture`

3.3.7 Multiple Stop Templates

A session type can have multiple stop templates, each classifying a different outcome:

```
<!-- HTTP 2xx → success -->
<tcp-message session-type="http-session"
             session-stop="true"
             session-result="success::2xx">
  <opcode match="^(2\d{2})"
          replace="HTTP $1">
    http.response.code
  </opcode>
</tcp-message>

<!-- HTTP 4xx → failure -->
<tcp-message session-type="http-session"
             session-stop="true"
             session-result="failure::4xx">
  <opcode match="^(4\d{2})"
          replace="HTTP $1">
    http.response.code
  </opcode>
</tcp-message>
```

The tutorial defines two independent session types:

Session Type	Key	Start	Stop (success)	Stop (failure)
`tcp-session`	IP 4-tuple	TCP SYN	TCP FIN	TCP RST
`http-session`	`tcp.stream` [2]	HTTP request	HTTP 2xx/3xx	HTTP 4xx/5xx

3.3.8 Output Directory Structure

Sessions are organized inside a subdirectory named after the input capture (its *stem* — the filename without extension), then by session type and outcome category:

```
output/
  tcp-sack-video/
    tcp-session/
      success/            ← e.g. `success::graceful-fin`
        00001_viewer.html
      failure/            ← e.g. `failure::tcp-reset`
        00001_viewer.html
      incomplete/         ← e.g. `incomplete::no-close`
        00001_viewer.html
    http-session/
```

[2] The Wireshark TCP conversation index. The tutorial uses this single-field key for `http-session` to show the simplest possible shape; switch to a 4-tuple when you need participant axes derived directly from the key, or when composing with `http2.streamid` — see Section 8.

```
      success/            ← e.g. `success::2xx`
        00001_viewer.html
      failure/            ← e.g. `failure::4xx`
        00001_viewer.html
```

This `<pcap-stem>/` subdirectory is always created, whether you pass a single `--input` file, multiple independent inputs, or use `--merge-inputs`. The consistent structure makes it safe to reuse the same `--output` directory across runs.

3.4 Part 3: Hosts File — Friendly Names

The `hosts.txt` file maps raw IP addresses to human-readable names. Without it, diagram axes show `10.0.0.145` and `186.15.230.24`. With it, they show `Firefox` and `Apache` grouped under `Client` and `Server`.

Open the `hosts.txt` file in any text editor and follow the inline comments as you read this section.

Use the `--hosts-file` flag to apply the hosts file when generating diagrams:

```
visualether generate \
  --fxt   explore.fxt.xml \
  --input tcp-sack-video.pcapng \
  --hosts-file hosts.txt \
  --output output/explore-named
```

3.4.1 Basic Format

Each line maps an IP address to a display name:

```
10.0.0.145     Firefox.Client
186.15.230.24  Apache.Server
```

The dot notation creates a two-level hierarchy: `Entity.Domain`. In the diagram, `Firefox` appears as the axis label under the `Client` domain header. Entities that share the same domain are grouped together and placed side by side — hierarchy controls both labeling and axis ordering.

3.4.2 Comments

Lines starting with `#` are ignored. Inline comments are also supported — a `#` token after the hostname strips the rest of the line:

```
# Full-line comment
10.0.0.145     Firefox.Client       # inline comment
186.15.230.24  Apache.Server        # another comment
```

3.4.3 Port-Specific Mapping

When using `--axis port`, you can map individual `IP:port` pairs:

```
# IPv4 with port
192.168.1.10:80    WebServer.Services
192.168.1.10:443   TLSProxy.Services

# IPv6 with port (bracket notation, per RFC 3986)
[2001:db8::1]:53   DNS.CoreNetwork
[2001:db8::2]:443  TLSServer.DMZ

# IPv6 without port (no brackets needed)
2001:db8::1        DNS.CoreNetwork
```

If a packet's `IP:port` has no port-specific entry, VisualEther falls back to the IP-only mapping.

3.4.4 Axis Coalescing

Map **multiple addresses to the same name** to coalesce their traffic onto a single diagram axis:

```
# IP coalescing: load-balanced servers appear as one axis
192.168.1.10  WebServer.Farm
192.168.1.11  WebServer.Farm
192.168.1.12  WebServer.Farm

# Dual-stack coalescing: IPv4 + IPv6 on the same axis
10.0.0.5          MyServer.Network
2001:db8::5       MyServer.Network

# Port coalescing (with --axis port): merge ephemeral ports
10.0.0.145:3372   Firefox.Client
10.0.0.145:3373   Firefox.Client
10.0.0.145:3374   Firefox.Client
```

> **Tip**
>
> Use `--axis ip` (the default) for most captures. Switch to `--axis port` when you need to distinguish multiple services on the same host — for example, localhost/loopback traffic where all traffic shares the same IP.

3.5 Next Steps

The tutorial covers the most commonly used FXT features. To go deeper:

- **Go deeper into FXT structure** — Section 5 covers the complete FXT document structure, Section 6 explains message element types and param forms, and Section 11 covers template groups, inheritance (`message-base` / `extends`), and priority
- **Learn session design** — Section 8 covers session type attributes, qualification, and key strategies, and Section 9 explains `auto-start` , `max-duration` , timeout handling, and full lifecycle rules
- **Apply the workflow to new protocols** — Section 2 shows how to use the `analyze` command workflow to discover fields and build templates for new captures
- **Study real-world examples** — Section 17 points to the Nick Russo PCAP Diagrams collection. After working through this tutorial, reading a few of those templates is one of the fastest ways to see idiomatic FXT applied at scale

4 Showcase: 5G NR Radio

Mobile-network radio captures are among the hardest surfaces to debug: a single MAC PDU on the wire unfolds into a five-layer stack (MAC → RLC → PDCP → RRC → NAS), with no IP or hostnames to anchor on and Wireshark shownames that don't fit on a sequence-diagram arrow. This chapter is a **guided tour** of the bundled `5g-nr-radio` sample, showing what the FXT primitives from Section 3 produce when applied to a layered radio-stack capture. It is the worked example of a general pattern: Section 13 explains how to recognize and decode **any** radio capture — the `DLT_USER` and UDP-envelope framings and the per-layer session keys — and Section 16.4 is the troubleshooting reference with the full 4G/5G role × stack matrix.

4.1 Setting Up

Bootstrap a project from the sample, then `cd` in:

```
visualether new 5g-nr-radio
cd 5g-nr-radio
```

The project ships three real UE-side air-interface PCAPs (Motorola Edge 30 Pro, Google Pixel 6a, SIMCom SIM8262E-M2 modem), an `explore.fxt.xml`, a `sessions.fxt.xml`, a `hosts.txt`, and a `visualether.toml` pre-loaded with the `tshark-args` needed to decode the DLT_USER1 (149) framing — so `visualether generate` works in this directory with no extra flags or per-machine `user_dlts` setup. The captures are sourced from the ocudu_docs `cots_ues` knowledge base (BSD-3-Clause-Open-MPI; see the SPDX sidecars next to the PCAPs).

These are **air-interface** captures: no IP, no Ethernet, no transport layer. The MAC PDU itself is on the wire (DLT_USER1 = 149), wrapped in a synthetic UDP envelope that lets Wireshark's `mac_nr_udp` heuristic dissector pick it up. With the heuristic enabled, the dissector unfolds the full 5G NR stack inside one frame: `mac-nr` → `rlc-nr` → `pdcp-nr` → `nr-rrc`, and (where the inner NAS isn't ciphered) → `nas-5gs`. The same DLT=149 envelope is also used by gNB-side `rlc-nr-pcap` and `pdcp-nr-pcap` outputs (bare RLC / PDCP PDUs with no MAC layer above); the project skeleton enables `rlc_nr_udp` and `pdcp_nr_udp` alongside `mac_nr_udp` so the same `visualether.toml` works for all three pcap modes. (For DLT_USER troubleshooting and the full 4G/5G role × stack table, see Section 16.4 — not required to read along.)

4.2 What the Sample Demonstrates

A handful of FXT features compose to turn that decode tree into something readable:

Feature	What the Sample Shows
Layered priorities	`explore.fxt.xml` puts NR-RRC procedure templates at priority 410-450, RLC AM Status PDUs at 350, PDCP PDUs at 280, MAC control

	elements at 200-250, generic LCID fallbacks at 100. Frames decode as the most informative layer they reach.
Multi-match	`sessions.fxt.xml` declares `multi-match="true"` (see Section 5.2) and eight session types. Five track the radio and control layers and populate on every capture — `rrc-connection`, `pdcp-bearer`, `rlc-bearer`, `mac-ue`, `nas-5gs` — and one MAC-NR frame contributes to all five in parallel (see Section 5.2.1 for how the layers map to session types). The other three — `sdap-bearer`, `ip-flow`, `sip` — cover the decrypted user plane and appear only once you supply the UE keys (Section 4.7).
Regex-replace addresses	There is no IP layer to derive endpoints from, so the templates use `replace` on `mac-nr.direction` to synthesize fixed `gNB` and `UE` axis labels.
Emoji-tagged opcodes + lightbulb remarks	Each `replace` emits an emoji prefix matched to the message's role (`RRC Setup Request`, `Security Mode Command`, `RLC AM Status PDU`, `Short BSR`, `Contention Resolution`). Every template carries a `[Frame N] ...` remark explaining the message's role per the relevant 3GPP spec — hover-over context, no separate cheat sheet needed.

4.3 Generating the Diagrams

> **Warning**
>
> Session tracking is a Professional / Server feature. On Community Edition, `sessions.fxt.xml` is skipped — the explore diagram still renders, but the per-layer session views below are not produced.

From inside the project directory, render both views for one of the captures:

```
visualether generate --input moto_edge_30_pro.pcap --serve
```

The `visualether.toml` written by `new` already carries the DLT_USER UAT mapping plus the `--enable-heuristic mac_nr_udp / rlc_nr_udp / pdcp_nr_udp` arguments in a `[defaults]` block — inherited by every tshark-driven subcommand (`generate`, `analyze`, `endpoints`, …) so no per-machine `user_dlts` setup or extra `--tshark-arg` flags are needed. The same `[defaults]` block also sets `nas-5gs.null_decipher:TRUE` because these test-rig captures negotiate the null NAS ciphering algorithm (NEA0); without that preference, the `nas-5gs` dissector treats post-SMC NAS messages as opaque ciphered envelopes even though they are on-the-wire plaintext. With it enabled, Registration Complete, PDU Session Establishment Request/Accept, Configuration Update Command, and 5GMM/5GSM cause codes surface in the diagram instead of appearing only as a sequence number

on a `Ciphered NAS` line. `generate` auto-iterates over `explore.fxt.xml` and `sessions.fxt.xml`, and `--serve` opens the Capture Atlas (Figure 3) on the rendered output. The sample README still includes copy-paste shell variants for users running outside a project directory.

> **Warning**
>
> `nas-5gs.null_decipher:TRUE` tells the dissector to parse post-Security-Mode-Complete NAS bodies as plaintext **unconditionally** — it does not check the negotiated cipher. If you point a project bootstrapped from this sample at a capture that uses real ciphering (NEA1 / NEA2 / NEA3), the dissector will mis-parse the ciphertext as 5GMM/5GSM IEs and surface plausible-looking but completely fabricated `message_type` and `cause` values. Confirm NEA0 from the (plaintext) Security Mode Command frame before reusing this `visualether.toml` for non-test captures; otherwise remove the `-o nas-5gs.null_decipher:TRUE` entry from `[defaults]` and accept that post-SMC NAS appears as `Ciphered NAS` envelopes.

4.4 Reading the Explore Diagram

The priority stack means each frame is labeled at the most informative layer it reaches:

Frame Type	Rendered As
PRACH preamble response	`Contention Resolution` (gNB→UE)
Msg3 (RRC Setup Request)	`RRC Setup Request` (UE→gNB)
Msg4 (RRC Setup)	`RRC Setup` (gNB→UE)
Msg5 (RRC Setup Complete + initial NAS)	`RRC Setup Complete` — with the NAS Registration Request piggyback surfaced as an inline param
Security Mode Command/ Complete	`Security Mode Command` / `Security Mode Complete`
RRC Reconfiguration adding DRBs	`RRC Reconfiguration` — with SRB/DRB IDs, t-PollRetransmit, BSR/ PHR timers, ROHC profile, SDAP header, etc. as params
RLC AM ACK feedback	`RLC AM Status PDU` — ACK_SN and any NACK list
Bare PDCP PDU (post-Security)	`UL PDCP SN=N` / `DL PDCP SN=N` — bearer ID, plane, MAC-I; inner content is ciphered (decrypt it with UE keys — see Section 4.7)

Buffer Status / Power Headroom	`Short BSR` / `Single-Entry PHR`
Generic MAC PDU (no higher-layer match)	`DL LCID n (name)` / `UL LCID n (name)` fallback

Bookmarked state transitions (RRC Setup Request, Setup, Setup Complete, Security Mode, Reconfiguration, Release, RAR, Contention Resolution, NAS Registration, Authentication) appear in the viewer's bookmark sidebar for jump-to navigation.

4.5 Reading the Session Diagrams

`sessions.fxt.xml` produces five parallel views from the same frames. Under `output/moto_edge_30_pro/` :

Path	What it contains
`rrc-connection/incomplete/00001_viewer.html`	Per-UE RRC procedure flow
`pdcp-bearer/incomplete/00001_viewer.html`	SRB1
`pdcp-bearer/incomplete/00002_viewer.html`	SRB2
`pdcp-bearer/incomplete/00003_viewer.html`	DRB1 (added on Reconfiguration)
`rlc-bearer/incomplete/00001_viewer.html`	Per-bearer ARQ
`mac-ue/incomplete/00001_viewer.html`	MAC CEs + LCID fallback
`nas-5gs/incomplete/00001_viewer.html`	Registration / Authentication / Service flow
`index.html`	Session Navigator

Each session is a different **view** of the same wire data, answering a different debugging question:

- `rrc-connection` — Did the RRC procedure complete? (Setup Request → Setup → Setup Complete → Security Mode → Reconfiguration → Release without per-PDU clutter)
- `pdcp-bearer/SRB1` — Did SRB1 see retransmissions? Duplicate PDUs are flagged inline with a `PDCP retx` marker.
- `rlc-bearer/SRB1` — Did the RLC AM Status PDUs arrive in time, or did `t-PollRetransmit` fire? (AMD PDUs and Status PDUs interleaved in air-time order.) Each RLC retransmission is bookmarked, so you can jump straight to it from the bookmark list.
- `mac-ue` — Was the UE granted resources for Msg3, and what's the BSR/PHR cadence?
- `nas-5gs` — Did NAS Registration succeed? (piggybacked Registration Request → Authentication → Security Mode → Registration Accept, free of the RRC scaffolding)

End-of-capture sessions are labeled `incomplete::mid-capture` because air-interface dumps rarely carry a clean teardown; see Section 9 for outcome-classification guidance.

4.6 gNB-Side RLC / MAC Captures

Everything so far has come from the three bundled UE-side MAC-NR captures, but the project skeleton is wired to decode more than just those. The same DLT=149 envelope is also produced by gNB-side `rlc-nr-pcap` and `pdcp-nr-pcap` outputs — bare RLC-NR or PDCP-NR PDUs with no MAC layer above — which is exactly why Section 4.1 enables `rlc_nr_udp` and `pdcp_nr_udp` alongside `mac_nr_udp`. With those heuristics already in place, a gNB capture drops straight into the same project.

To try it on real gNB output, download the two example captures attached to Wireshark MR !12834 (by srsRAN Project contributor Robert Falkenberg):

- `gnb_rlc.pcap` — bare RLC-NR PDUs from a srsRAN_Project gNB (DLT=149 envelope, `rlc-nr` tag — needs `rlc_nr_udp`)
- `gnb_mac.pcap` — MAC-NR PDUs from the same gNB (DLT=149 envelope, `mac-nr` tag — needs `mac_nr_udp`)

Drop either into a project bootstrapped from this sample and `visualether generate --input gnb_rlc.pcap` decodes it with the existing `visualether.toml` — no extra flags, because the three `*_nr_udp` heuristics are already in the inherited `[defaults]` block.

4.7 Decrypting the User Plane (SDAP + IP)

A gNB capture exercises the same radio layers from the other end; decryption opens up a layer the bundled captures keep hidden altogether. They are ciphered on the user plane (NEA2 / NIA2), so post-Security DRB PDUs render as bare 🔼 `UL PDCP SN=N` with the inner SDAP and IP opaque. When a capture's UE security keys are available — for example Wireshark work item 19757's `r16_mdt_nr_mac.pcap`, which publishes them — VisualEther can decipher the user plane so the SDAP QoS-flow header and inner IPv6 dissect.

The decrypt recipe lives in `visualether.toml` — the `pdcp_nr_ue_keys` UAT row (UEId plus the RRC and user-plane cipher keys), `pdcp-nr.decipher_userplane:TRUE`, `mac-nr.lcid_to_drb_mapping_source:"From configuration protocol"`, and `rlc-nr.call_pdcp_for_{ul,dl}_drb` at the PDCP SN length (default 18-bit). This sample ships a ready-made `r16-mdt-19757.visualether.toml` carrying exactly that recipe for the work-item-19757 capture — copy it over `visualether.toml` and substitute your own UEId and keys:

```
[defaults]
tshark-arg = [
    "--enable-heuristic", "mac_nr_udp",
    "--enable-heuristic", "rlc_nr_udp",
    "--enable-heuristic", "pdcp_nr_udp",
    "-o", "pdcp-nr.decipher_userplane:TRUE",
    "-o", "mac-nr.lcid_to_drb_mapping_source:From configuration protocol",
```

```
    "-o", "rlc-nr.call_pdcp_for_ul_drb:18-bit SN",
    "-o", "rlc-nr.call_pdcp_for_dl_drb:18-bit SN",
    "-o", "uat:pdcp_nr_ue_keys:\"1\",\"95F23667A60D91689B31772783E6FDDD\",
\"4CB12A7FB40AE866AE59FC9529E0BC5B\",\"\",\"\"",
]
```

Once decrypted, the `explore.fxt.xml` SDAP templates (priority 360, above the RLC templates) label each DRB user-plane PDU by its QoS flow — 🔼 `UL SDAP → QFI 2` — with the inner IPv6 / TCP / ESP dissected instead of an opaque ciphered envelope. 4G LTE is identical via the `pdcp_lte_ue_keys` UAT (LTE has no SDAP layer, so the inner IP dissects directly under PDCP-LTE).

Decryption also adds three user-plane views to the session navigator, layered above the radio sessions. `sdap-bearer` groups each DRB's traffic by QoS flow; `ip-flow` shows the inner end-user IP exchange as a host-to-host conversation (here an IPv6 flow); and `sip` reconstructs the IMS signalling carried inside it — the work-item-19757 capture contains a SIP `REGISTER` answered with `401 Unauthorized`. Inner-IP milestones such as the TCP connection opening and closing are called out with 🚩 `Expert` annotations drawn from Wireshark's own expert info.

> **Tip**
>
> The work-item-19757 capture is not redistributable, so it isn't bundled. Download `r16_mdt_nr_mac.pcap` from the work item into the project directory, then `cp r16-mdt-19757.visualether.toml visualether.toml` and run `visualether generate` — the user plane decrypts and SDAP renders with no further flags.

If you are driving the analysis from Claude Code rather than editing the toml by hand, the `materialize_config` MCP tool assembles this same recipe from a `pdcp_nr_keys` / `pdcp_lte_keys` list, then cross-checks the keys against the capture (cipher algorithm, UE ids) and reports how many user-plane frames actually decrypted — so a missing or wrong key surfaces immediately. See Section 15.

4.8 Adapting the Pattern to Your Own Captures

The per-layer session-type pattern (one session per protocol layer, all keyed off a UE identifier) generalizes to any wire format that provides a nested decode tree. VisualEther ships ready-to-use samples for the most common mobile workflows:

Sample	Use Case
`4g-lte`	Same five-layer pattern as `5g-nr-radio` but for LTE: `mac-lte` / `rlc-lte` / `pdcp-lte` / `lte-rrc` / `nas-eps`. Start here if your capture is LTE rather than NR.
`4g-attach`, `5g-attach`	Initial UE attach procedures end-to-end — RRC + NAS + S1AP/NGAP — correlated as a single session.

`4g-volte-sms`, `ims`	VoLTE call setup and SMS over IMS, with SIP / Diameter / RTP correlated by call-ID and IMSI.
`5g-core-establishment` `5g-srsran-ngap`	NGAP + NAS-5GS over SCTP for 5G core signaling — split into `ngap-ue`, `nas-5gs`, and `sctp-assoc` so each stack layer gets its own lifecycle view.
`gtp`, `4g-gx-gy`, `4g-lte-epc-flows`	User-plane (GTP-U) and policy-plane (Diameter Gx/Gy) flows for EPC and 5GC interfaces.

The general recipe: identify the natural *correlator* at each layer (RNTI, bearer ID, IMSI, stream ID), declare a session type per layer keyed on that field, tag every message template with the appropriate `session-type`, and set `multi-match="true"` at the root. The same captured frames will then appear as independent per-layer session views.

4.9 Next Steps

The FXT chapters that follow (Section 5 through Section 11) cover the building blocks this showcase uses — session types, qualifiers, multi-match, priority, and inheritance. Section 13 walks through applying the same recipe to a protocol that doesn't ship with a sample.

5 FXT Overview

The Introduction covers FXT (Field Extraction Template) conceptually. This chapter covers the file format itself — the two flavors (**explore** and **sessions**), the document structure, and the matching modes.

> **Tip**
>
> Before reading this chapter in detail, skim a few real-world templates to see what idiomatic FXT looks like end to end. Section 17 points to the Nick Russo PCAP Diagrams collection, with hand-tuned `explore.fxt.xml` files illustrating every attribute covered below.

5.1 Document Structure

Every FXT file has the same root structure:

```
<?xml version="1.0" encoding="utf-8" ?>
<FXT>
  <!-- Optional: CI/CD session filtering -->
  <session-filter outcome-in="failure incomplete" />

  <!-- Optional: Session type definitions -->
  <session-type name="my-session" ...>...</session-type>

  <!-- Required: Message template definitions -->
  <tcp-message>...</tcp-message>
  <udp-message>...</udp-message>
</FXT>
```

The root tag is case-insensitive: `<FXT>` and `<fxt>` are both accepted (the closing tag must match the opening tag, per XML rules). Built-in samples use uppercase `<FXT>`.

5.1.1 Element Order

1. `<session-filter>` — optional, at most one
2. `<session-type>` — optional, zero or more
3. `<template-group>` — optional, zero or more
4. Message elements — required, one or more

5.1.2 Minimal Example

```
<?xml version="1.0" encoding="utf-8" ?>
<FXT>
  <tcp-message style="forest-green">
```

```
    <opcode>http.request.method</opcode>
    <param>http.request.uri</param>
  </tcp-message>
</FXT>
```

One root attribute changes how templates are matched against each packet — `multi-match`, which defaults to `false`. The two sections below cover the default single-match behaviour and the multi-match alternative.

5.2 Multi-Template Matching

By default, FXT uses **first-match-wins**: only the first matching template generates a message for each packet. The tutorial in Section 3.3 introduces the basic switch (`<FXT multi-match="true">`) and shows it in the HTTP-over-TCP session example. The mechanics that matter when you scale this up:

- Each matching template updates its own session type independently — a single packet can contribute to multiple sessions.
- Priority still applies, so higher-priority templates are still checked first.
- The first match includes the timestamp; subsequent matches appear as sub-messages on the same arrow.
- The first match also owns the arrow's **bookmark** flag and primary label — later matches contribute only sub-messages, so their `bookmark="true"` has no effect.
- A `<param required="true">` gate drops only its own template; the other matching templates still fire (Section 7.6.1).

The worked example below traces what this looks like on a single 5G NR frame that decodes through five protocol layers.

> **Important**
>
> `multi-match` slices a packet across **templates** (each template feeding a different session-type). For multiplexed protocols where a **single** template's session key has several values in one packet (e.g., HTTP/2 streams sharing one TCP segment), the relevant mechanism is multi-value key field splitting — see Section 8.

> **Important**
>
> Because the first match owns the bookmark, order a `bookmark="true"` template ahead of any unbookmarked catch-all that could match the same frame, or the bookmark is silently lost.

5.2.1 Mapping Protocol Layers to Sessions

The 5G NR Radio showcase (Section 4) is the canonical example of where `multi-match` earns its keep. A single MAC-NR frame decodes through the whole radio stack — MAC → RLC → PDCP → RRC → NAS, and after decryption SDAP → IP → SIP — so its `sessions.fxt.xml` declares one session type per layer and sets `multi-match="true"` at the root:

- `rrc-connection` and `nas-5gs` — the control-plane procedures (RRC Setup / Reconfiguration / Release, NAS Registration / Authentication).
- `pdcp-bearer`, `rlc-bearer`, and `mac-ue` — the per-bearer and per-UE radio layers (PDCP SN, RLC ARQ, MAC control elements).
- `sdap-bearer`, `ip-flow`, and `sip` — the user plane, populated only once the UE keys decrypt it (Section 4.7).

Each layer's template carries the matching `session-type`, so one physical frame is added to every session whose layer it touches. Without `multi-match`, only the highest-priority template would fire — a signalling frame that decoded all the way to NAS would land in `nas-5gs` alone, and the `rrc-connection`, `pdcp-bearer`, and `rlc-bearer` views would silently lose it. Multi-match is what keeps each per-layer view complete. See Section 4 for the diagrams these session types produce.

For **phase-based** sub-sessions on a single layer (e.g., PAP authentication vs IPCP negotiation on the same PPPoE link), see Section 8.8 in Section 8.

5.3 Single-Match Template Selection

In default (single-match) mode, every packet matches at most one template. Selection is governed entirely by two rules:

1. **Priority desc** — templates with a higher `priority="N"` attribute are checked first.
2. **File order asc** — within the same priority (including the default priority of `0`), the template listed earlier in the FXT wins.

There is no automatic protocol-layer inference. The engine does not know that HTTP is "higher" than TCP. If you want a higher-layer label to win over a lower-layer fallback, say so explicitly: either place the higher-layer template earlier in the file, or give it a higher `priority`. The tutorial in Section 3.2 demonstrates the file-order approach on a TCP/HTTP capture — HTTP templates listed before the TCP catch-all win for HTTP packets and fall through to TCP for everything else.

Use explicit `priority` when file order alone won't do — most importantly, when a loss or error template must outrank an earlier normal-traffic template at the same priority. The HTTP/3 sample's QUIC retransmission template at `priority="210"` outranks normal HTTP/3 frames at `priority="200"` for exactly this reason.

Fall-through is not limited to the `<opcode>`. A template can require a **second** field to match before it fires, via a `<param required="true" match="…">` (Section 7.6.1): when that param doesn't match, the template is skipped during selection and the engine continues to the next one, just as a non-matching opcode would. This is how templates that share an opcode field are told apart — for example per-transport session types gated on `tcp.port`.

> **Tip**
>
> To reduce repetition across templates, see Section 11 for template groups, template inheritance (`message-base` / `extends`), and priority.

5.4 Next Steps

With the document shape, root attributes, and matching modes in place, the FXT chapters that follow cover the moving parts in detail — message elements in Section 6, field extraction in Section 7, and the session machinery in Section 8 and Section 9.

6 Message Templates

Message templates define which protocol messages VisualEther extracts from a packet capture and how they appear in the sequence diagram.

6.1 Message Types by Transport

Each message element corresponds to a transport protocol. VisualEther automatically determines the source and destination addresses from the transport type.

Element	Transport	Example Protocols
`<tcp-message>`	TCP/IPv4	HTTP, BGP, LDAP, TLS, Modbus
`<tcpv6-message>`	TCP/IPv6	IPv6 TCP variants
`<udp-message>`	UDP/IPv4	DNS, SIP, GTP, NTP, QUIC
`<udpv6-message>`	UDP/IPv6	IPv6 UDP variants
`<sctp-message>`	SCTP/IPv4	Diameter, S1AP, NGAP, ISUP
`<sctpv6-message>`	SCTP/IPv6	IPv6 SCTP variants
`<ip-message>`	IP layer	ICMP, raw IP
`<ipv6-message>`	IPv6 layer	ICMPv6
`<mac-message>`	MAC layer	Ethernet frames (ARP, LLDP, STP, IS-IS)
`<wifi-message>`	WiFi	802.11 frames
`<message>`	Generic	Non-IP protocols with explicit source/destination

6.1.1 Dual-Stack IPv4/IPv6 (`auto-v6="true"`)

Transport-typed elements only match their declared IP version: `<tcp-message>` reads `ip.src` / `ip.dst` and matches IPv4 packets, `<tcpv6-message>` reads `ipv6.src` / `ipv6.dst` and matches IPv6. An IPv4-only FXT silently drops every IPv6 packet on the floor.

The shortest route to dual-stack coverage is the root attribute `auto-v6="true"` :

```
<FXT auto-v6="true">
  <!-- Author writes IPv4 once; loader synthesises the tcpv6-message
       twin automatically and inserts it immediately after. -->
  <tcp-message>
    <opcode>http.request.method</opcode>
```

```
  </tcp-message>
</FXT>
```

Equivalent to writing both templates by hand, but without the maintenance burden of keeping the pair in sync. The same expansion applies to `<udp-message>` -> `<udpv6-message>` and `<sctp-message>` -> `<sctpv6-message>`. Generic `<message>` and `<ip-message>` / `<ipv6-message>` are not auto-twinned — generic messages have explicit source/destination, so there's nothing to derive.

Explicit override. An explicit `<tcpv6-message>` (or v6 variant of any transport) with the same opcode field and `@match` as an IPv4 sibling **suppresses** the auto-twin for that template. Use this when the IPv6 variant genuinely needs different style, params, or remarks:

```
<FXT auto-v6="true">
  <tcp-message style="forest-green">
    <opcode match="^GET$">http.request.method</opcode>
  </tcp-message>

  <!-- This wins over the auto-twin; auto-twin is not generated for GET. -->
  <tcpv6-message style="ocean-blue">
    <opcode match="^GET$">http.request.method</opcode>
  </tcpv6-message>
</FXT>
```

When `auto-v6` is omitted (the default), the FXT load-time validator warns whenever the FXT mixes both transport variants with asymmetric coverage — write the missing twins, or set `auto-v6="true"` and have the loader do it.

The `FXT-TWIN` coverage check has two refinements for protocols that don't fit the simple "same fields on both IP versions" model — sibling protocols whose v4 and v6 forms use entirely different Wireshark field names (`dhcp` ↔ `dhcpv6` , `icmp` ↔ `icmpv6` , and the like), and protocols that exist on only one IP version (`single-stack="true"`). It also needs `addr.*` -keyed sessions so the synthetic IPv6 twins can form session keys. See Section 11.3 for the full rules.

6.2 Message Attributes

Attribute	Type	Default	Description
`style`	string	none	Color and line style
`filter`	boolean	false	Deduplicate by (source, destination, opcode) — keeps the first occurrence per direction and suppresses repeats. The kept message retains full params and remarks. Use for high-volume frames (Beacons, QoS Data, RTP, ACK) to reduce diagram clutter while preserving one representative instance

`bookmark`	boolean	false	Create a PDF bookmark for this message
`priority`	integer	0	Template matching priority. Higher = checked first
`session-type`	string	none	Reference to a `<session-type>` definition
`session-start`	boolean	false	Marks the beginning of a session
`session-stop`	boolean	false	Marks the end of a session
`session-result`	string	none	Outcome classification on session stop
`extends`	string	none	Inherit params/remark from a `<message-base>` (see Section 11)
`single-stack`	boolean	false	Opt out of the `FXT-TWIN` dual-stack coverage check and `auto-v6="true"` expansion. Use on transport templates for protocols that are single-stack by design (NBNS, NetFlow v5). See Section 11.3 for the full rules

6.3 Template Matching

Templates use **first-match-wins** by default (see Section 5.2 for the full mechanics and the `multi-match` alternative). For each packet, the first message element whose `<opcode>` field matches wins, so place specific patterns before generic fallbacks.

```
<!-- Specific: matches only BGP OPEN -->
<tcp-message style="royal-blue">
  <opcode match="^Type: (OPEN) Message">bgp.type</opcode>
</tcp-message>

<!-- Generic: matches any BGP message -->
<tcp-message style="steel-blue">
  <opcode match="^Type:">bgp.type</opcode>
</tcp-message>

<!-- Fallback: matches any TCP packet -->
<tcp-message style="slate dotted">
  <opcode match=".*Flags:.*\((.*)\)"
          replace="TCP ($1)">tcp.flags</opcode>
</tcp-message>
```

6.4 Priority Scoring

The `priority` attribute overrides file-order matching:

- **Higher priority = checked first**, regardless of position in the file
- Equal-priority templates retain file order (stable sort)
- The default priority is 0
- Negative priorities create fallback templates

```
<!-- Checked first (priority 100) -->
<udp-message priority="100" style="royal-blue">
  <opcode>http3.frame_type</opcode>
</udp-message>

<!-- Checked second (priority 10) -->
<udp-message priority="10" style="slate">
  <opcode>quic.packet_type</opcode>
</udp-message>

<!-- Checked last (negative priority) -->
<udp-message priority="-10" style="graphite">
  <opcode>udp.length</opcode>
</udp-message>
```

> **Tip**
>
> For the common case of "prefer the higher-layer protocol when several templates match the same packet" (e.g., HTTP/3 over QUIC, HTTP over TCP), either list the higher-layer template first in the file (first-match-wins) or assign it a higher `priority`. Use a negative `priority` to push a fallback template below everything else.

6.5 Protocol Message Structure

Protocol-specific message elements take this shape:

```
<tcp-message style="forest-green" session-type="my-session">
  <opcode>field.name</opcode>        <!-- Required: message type -->
  <param>field.name</param>          <!-- Optional: 0 or more parameters -->
  <remark>field.name</remark>        <!-- Optional: annotation -->
</tcp-message>
```

6.5.1 Child Element Ordering

Children of a message template must appear in this order, and siblings of the same type must remain grouped contiguously:

1. `<opcode>` — required, exactly one.
2. `<param>` — any number of params, all contiguous. No other element may appear between two `<param>` elements.
3. `<source>` / `<destination>` — only on generic `<message>` (see below).
4. `<remark>` — **at most one** per template.

Violating this order — for example, placing a `<param>` after a `<remark>` or using two `<remark>` elements in one template — produces a parse error such as:

```
Failed to parse FXT XML: duplicate field `param` at line 19 column 79
Failed to parse FXT XML: duplicate field `remark` at line 15 column 32
```

A template can carry any number of `<param>` elements but only one `<remark>`, and the params must all come before the remark. If you need to surface two annotation values, keep one as a `<remark>` and promote the other to a `<param>`:

```
<!-- Wrong: two <remark> elements -->
<ip-message>
  <opcode>icmp.type</opcode>
  <remark>icmp</remark>
  <remark>icmp.resptime</remark>
</ip-message>

<!-- Right: promote the extra annotation to a <param> -->
<ip-message>
  <opcode>icmp.type</opcode>
  <param match="(.+)" replace="⏱ Response time: $1 ms">icmp.resptime</param>
  <remark match=".*Frame Number: (\d+)" replace="💡 [Frame $1] ICMP ping
verifies reachability">frame.number</remark>
</ip-message>
```

6.6 Generic Message Structure

The generic `<message>` element requires explicit source and destination definitions. Use it for non-IP protocols where the transport-specific elements do not apply, such as IEEE 802.15.4 (Zigbee), Bluetooth, or CAN bus traffic. Example for an IEEE 802.15.4 (Zigbee) capture:

```
<message style="amethyst">
  <opcode>zbee_nwk.cmd.id</opcode>
  <param>zbee_nwk.cmd.link_status_count</param>
  <source>
    <address>wpan.src64</address>
  </source>
  <destination>
    <address>wpan.dst64</address>
```

```
  </destination>
  <remark match=".*Frame Number: (\d+)" replace="💡 [Frame $1] Zigbee NWK
command">frame.number</remark>
</message>
```

`<port>` is also supported under `<source>` / `<destination>` when you need explicit endpoint control on a port-bearing transport.

6.7 WiFi Message Templates

`<wifi-message>` resolves sequence diagram axes from 802.11 MAC addresses. It works for **all** 802.11 frame types — management, data, and control:

```
<!-- Management frame -->
<wifi-message bookmark="true" style="sea-green">
  <opcode match="Type/Subtype: Authentication (.*)"
          replace="Authentication $1">wlan.fc.type_subtype</opcode>
  <param>wlan.fixed.auth.alg</param>
  <param>wlan.fixed.status_code</param>
</wifi-message>

<!-- Control frame -->
<wifi-message style="sea-green dotted">
  <opcode match="Type/Subtype: (?:802\.11 )?Block Ack (.*)"
          replace="✅ Block Ack $1">wlan.fc.type_subtype</opcode>
  <param display="brief" match=".*TID.*: (0x\d+)"
         replace="🏷️ TID: $1">wlan.ba.basic.tidinfo</param>
</wifi-message>
```

tshark version note: Some `tshark` versions prefix control-frame subtypes with `802.11 ` (for example, `Type/Subtype: 802.11 Block Ack (0x0019)`). Use `(?:802\\.11 )?` in `match` to handle both variants.

6.7.1 WiFi Template Selection

`<wifi-message>` covers every 802.11 frame type — management, data, EAPOL handshake, and all control frames — and resolves axes automatically from `wlan.ta` (transmitter) and `wlan.ra` (receiver). The one special case is **CTS** and **ACK** control frames, which carry only `wlan.ra` and therefore appear as self-messages on the receiver's axis.

6.7.2 Constant Axis Labels for Point-to-Point Protocols

For non-Ethernet link types (PPP serial, ATM, Frame Relay) where no IP or MAC address fields exist, use `replace` on `<address>` to create constant axis labels. Both source and destination can watch the **same field** while producing different labels:

```
<!-- PPP serial capture — ppp.address is 0xFF in every packet -->
<message style="forest-green">
  <opcode match="Code: (.*)"
          replace="PPP $1">ppp.code</opcode>
  <param>ppp.protocol</param>
  <source>
    <address match=".*"
             replace="Local">ppp.address</address>
  </source>
  <destination>
    <address match=".*"
             replace="Remote">ppp.address</address>
  </destination>
</message>
```

Both `<address>` elements watch `ppp.address` (which is `0xFF` in every PPP packet). The `replace` transforms the matched value into “Local” for the source and “Remote” for the destination, creating meaningful sequence diagram axes. The 5G NR Radio showcase (Section 4) uses the same trick on `mac-nr.direction` to synthesize fixed 📡 `gNB` and 📱 `UE` axes for air-interface captures that have no IP layer.

When you ask Claude Code to analyze a capture on a non-Ethernet link type (PPP, ATM, Frame Relay, and similar), it spots the missing IP / MAC layer and suggests this constant-label pattern — so you can usually skip writing the `<source>` / `<destination>` block by hand.

6.8 Next Steps

This chapter covered which transport element to pick and the shape of each message template. Section 7 covers the `<opcode>`, `<param>`, and `<remark>` children in depth (match patterns, replace strings, name-value pairs). Section 10 covers the `style` attribute’s color palette and line shapes.

7 Field Extraction

Field extraction controls how VisualEther reads values from packet dissection fields and transforms them for display in a sequence diagram.

7.1 The `<opcode>` Element

The `<opcode>` element identifies the message type. It is required in every message template. Its field name refers to a Wireshark dissector field.

```
<!-- Simple: use the field value directly -->
<opcode>bgp.type</opcode>

<!-- With display mode -->
<opcode display="brief">http.request.method</opcode>

<!-- With regex transformation -->
<opcode
    match="^Type: (OPEN) Message \(.*\)"
    replace="BGP $1"
>bgp.type</opcode>
```

7.2 The `<param>` Element

The `<param>` element extracts additional fields, which are displayed as message parameters:

```
<!-- Simple parameter -->
<param>bgp.open.myas</param>

<!-- With transformation -->
<param
    match="(.*) \(.*\)"
    replace=". $1"
>bgp.nlri_prefix</param>
```

Each message template can have any number of `<param>` elements.

7.2.1 Name-Value Param Form

Some protocols expose related name/value data as separate fields. The name-value form combines them into a single `"name: value"` row:

```
<param>
  <name match="^Parameter name: (.+)$"
        replace="$1">pgsql.parameter_name</name>
  <value match="^Parameter value: (.+)$"
         replace="$1">pgsql.parameter_value</value>
</param>
```

Rules:

- The `<param>` element must have **no attributes** when using child elements
- Both `<name>` and `<value>` children are required
- Each child supports the standard field extraction attributes
- Fields are paired in order: each name is held until its corresponding value arrives

7.3 The `<remark>` Element

The `<remark>` element adds an annotation to the message in the diagram. It supports the same field extraction attributes as `<opcode>` (`display` , `match` , `replace` , `skip-inner`), but it does not support the `<name>` / `<value>` child form available in `<param>` .

```
<remark>frame.time_relative</remark>

<!-- With regex transformation -->
<remark match=".*: (.*)" replace="$1">frame.time_relative</remark>
```

7.4 Field Extraction Attributes

All field elements (`<opcode>` , `<param>` , and `<remark>`) support these attributes:

Attribute	Description	Example
`display`	Value only (`brief`) or full Wireshark text (`detailed` , default)	`display="brief"`
`match`	Regex pattern to match	`match="^Type: (.*)"`
`replace`	Replacement with capture groups	`replace="BGP $1"`
`skip-inner`	Skip N inner layers for encapsulated protocols	`skip-inner="1"`

`<param>` additionally supports `required="true"` , which gates the template: when the field is missing or the regex doesn't match, the message is dropped. Has no effect without `match` . See the SCTP ABORT example below.

7.5 Display Modes

Wireshark stores two representations for every field: a short raw value and a longer human-readable description. The `display` attribute selects which one appears in the diagram.

- **`detailed`** (default) — the full Wireshark text, typically a label followed by the value.
- **`brief`** — just the value itself, without the label prefix.

Mode	Description	Example Field	Output
`detailed`	Full Wireshark text (default)	`tcp.srcport`	`Source Port: 443`
`brief`	Value only	`tcp.srcport`	`443`

For example, with `bgp.open.holdtime`:

```
<!-- detailed (default): shows "Hold Time: 90" -->
<param>bgp.open.holdtime</param>

<!-- brief: shows just "90" -->
<param display="brief">bgp.open.holdtime</param>
```

For **opcodes**, `brief` is useful when the label is redundant or when you plan to apply a `replace` transformation to the value.

Use `display="brief"` on `<param>` with caution. Brief mode strips the human-readable label and keeps only the raw value. When combined with `replace`, the transformed result is split on the first colon for the name:value table. This is *safe* for:

- **Bitmap/numeric fields** where `brief` strips the bitfield prefix: `wlan.seq` (raw: `0`), `wlan.qos.tid` (raw: `0`), `scsi.lun` (raw: `0x0001`), `http2.streamid` (raw: `1`), `coap.type` (raw: `0`), `rtpevent.end_of_event` (raw: `True`)
- **HTTP/1.x fields** where the `detailed` showname has a redundant label prefix: `http.request.uri` (detailed: `Request URI: /path`, brief: `/path`)

It is *unsafe* for fields whose raw value contains colons — IPv6 addresses (`fe80::1`), MAC addresses (`00:0c:29:fc:2c:3b`), and similar fields where the colon would be misinterpreted as the name:value separator. For those fields, use the default `detailed` mode with `match` to extract the value from the showname.

It is also *safe* for `<address>` elements inside `<message>` templates — for example, `<address display="brief">wlan.ta</address>` returns the plain MAC address instead of `Transmitter address: 84:b2:61:24:a2:00`, preventing axis-splitting issues in WiFi captures (see Section 6).

7.6 Filtering and Transforming with Regex

`match` and `replace` provide full regex matching with capture groups for both filtering and transformation:

```
<!-- Include only SYN packets -->
<opcode match=".*SYN.*">tcp.flags</opcode>

<!-- Filter and reformat in one step -->
<opcode
    match=".*Flags:.*\((.*)\)"
    replace="TCP ($1)"
>tcp.flags</opcode>
```

For `<opcode>` elements, the template matches only packets whose field value matches `match` — a non-match drops the whole message. For `<param>` elements, the rule is different: a non-matching `match` silently omits the param, but the rest of the message still fires (because the opcode already matched). To make a `<param>` gate the template the same way an opcode does, add `required="true"`.

7.6.1 Gating templates by param value (`required="true"`)

A common pattern is to split a single opcode into multiple session-result outcomes based on a field value. For example, the SCTP ABORT chunk usually means failure, but RFC 4960 §3.3.10.12 cause 12 (“User Initiated ABORT”) signals an intentional teardown that should be classified as a success outcome.

`required="true"` on a `<param>` enables this: the regex both gates the template (drop the submessage when the field is absent or the regex doesn’t match) and transforms the captured value for display. Place the more-specific template first — first-match wins:

```
<!-- Match ABORT only when cause-code says "User initiated" -->
<sctp-message session-stop="true" session-result="success::user-initiated-
abort">
  <opcode match="Chunk type: (ABORT) \(6\)"
          replace="🛑 SCTP $1">sctp.chunk_type</opcode>
  <param required="true"
         match="^(?:[^:]+:\s*)?(.*User initiated.*)$"
         replace="❌ Cause: $1">sctp.cause_code</param>
</sctp-message>

<!-- Fallback: any other ABORT (or none with cause field) -->
<sctp-message session-stop="true" session-result="failure::abort">
  <opcode match="Chunk type: (ABORT) \(6\)"
          replace="🛑 SCTP $1">sctp.chunk_type</opcode>
  <param match="^(?:[^:]+:\s*)?(.+)$"
         replace="❌ Cause: $1">sctp.cause_code</param>
</sctp-message>
```

With `required="true"`, the gating regex doubles as the value formatter when paired with `replace`.

When the gate runs. `required="true"` is evaluated during template *selection*, not after it — the same phase that matches the `<opcode>`. In single-match mode this is exactly what makes the fallback above

work: when the first template's required param doesn't match, that template produces no message and the engine *falls through* to the next matching template, subject to the usual priority + file-order rules (Section 5.3). A required param is therefore a way to add a **second** match condition beyond the opcode. One practical use is separating session types that share an opcode field: gate each template with a `required="true"` param on a discriminating field — e.g. `tcp.port` — and a frame falls through to the template (and session type) for its own transport.

Under `multi-match="true"` the gate still drops its own template, but every other matching template fires independently — a failed `required` gate removes only that template's submessage, not the others (Section 5.2).

7.7 Encapsulated Protocol Layer Selection (`skip-inner`)

When working with tunneling protocols such as GTP-U, a single packet can contain multiple layers with the same field names (for example, `ip.src` in both the outer and inner headers). By default, VisualEther uses the **innermost** (last) occurrence.

Value	Behavior
`skip-inner="0"`	Use innermost layer (default)
`skip-inner="1"`	Skip 1 inner layer, use outer (for 2-layer encapsulation)
`skip-inner="N"`	Skip N inner layers

7.7.1 Example: GTP-U Tunnel Endpoints

```
<!-- Show OUTER tunnel endpoints (gNB <-> UPF) -->
<message style="teal">
  <opcode match="^Message Type:\s*T-PDU.*$"
          replace="GTP-U T-PDU">gtp.message</opcode>
  <param>gtp.teid</param>
  <source>
    <address skip-inner="1">ip.src</address>
    <port>udp.srcport</port>
  </source>
  <destination>
    <address skip-inner="1">ip.dst</address>
    <port>udp.dstport</port>
  </destination>
</message>
```

Without `skip-inner="1"`, the message would show UE – DN (the inner payload IPs) instead of the tunnel endpoints.

7.8 Regex Patterns Reference

7.8.1 Basic Syntax

Pattern	Meaning
`^`	Start of string
`$`	End of string
`.`	Any character
`.*`	Zero or more characters
`.+`	One or more characters
`\d`	Digit [0-9]
`\w`	Word character [a-zA-Z0-9_]
`\s`	Whitespace
`(...)`	Capture group
`(?<name>...)`	Named capture group

7.8.2 Replacement Syntax

Pattern	Meaning
`$1` , `$2`	Numbered capture groups
`${name}`	Named capture group
`$0`	Entire match

7.8.3 XML Character Escaping

In XML attributes, special characters must be escaped:

Character	Escape
`<`	`&lt;`
`>`	`&gt;`
`&`	`&amp;`
`"`	`&quot;`

7.8.4 Common Patterns

```
<!-- Extract from "Type: VALUE Message (number)" format -->
match="^Type: (.*) Message \(.*\)"
replace="$1"
<!-- Input: "Type: OPEN Message (1)" -> Output: "OPEN" -->

<!-- Extract from parentheses -->
match=".*Flags:.*\((.*)\)"
replace="TCP ($1)"
<!-- Input: "Flags: 0x012 (SYN, ACK)" -> Output: "TCP (SYN, ACK)" -->

<!-- Add prefix -->
match="(.*)"
replace=". $1"
<!-- Input: "My AS: 65033" -> Output: ". My AS: 65033" -->
```

7.9 Localization and Emoji Annotations

`replace` accepts arbitrary literal text alongside capture-group references — including translations and emoji — so the same PCAP can produce diagrams labeled in English, Japanese, Korean, Hindi, Arabic, or any other script just by writing different FXT files (Figure 5).

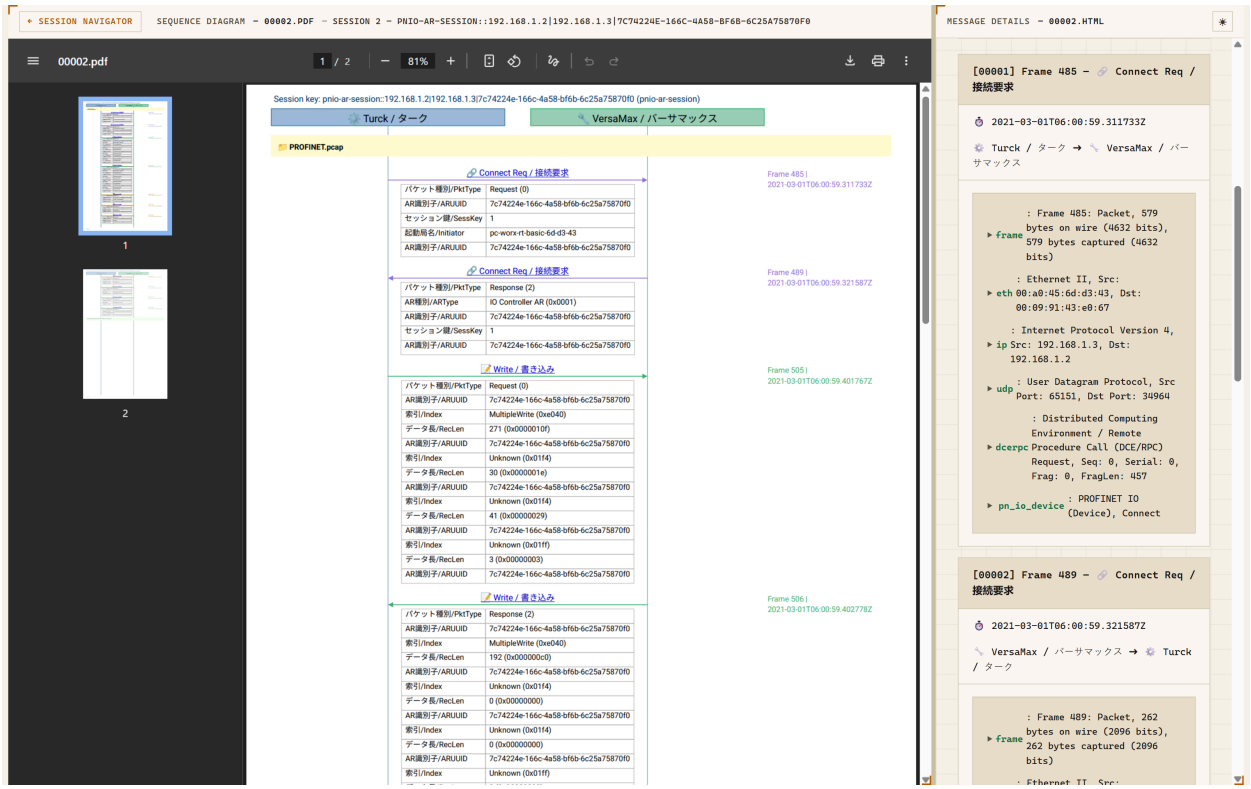

Figure 5: A PROFINET session diagram with bilingual labels (English + Japanese) and emoji. The axis headers show localized device names from the hosts file, while message opcodes and parameters use `replace` to display both languages.

7.9.1 Bilingual Labels with Emoji

A PROFINET opcode template producing an emoji + English + Japanese label:

```
<udp-message style="amethyst">
  <opcode match="Operation: Connect.*"
          replace="📎 Connect / 接続">pn_io.opnum</opcode>
</udp-message>
```

The same `match` / `replace` technique works on `<param>` elements to highlight key fields. With multiple params sharing the same tshark field name (e.g., `bgp.update.path_attribute`), the **first-match-wins** behavior lets you apply different emoji to different instances — place specific-regex params before a generic fallback so each field instance is consumed by exactly one param.

```
<ip-message style="forest-green">
  <opcode match="Message Type: (Hello Packet).*"
          replace="👋 OSPF $1">ospf.msg</opcode>
  <param match="Source OSPF Router: (.*)"
         replace="🆔 Source OSPF Router: $1">ospf.srcrouter</param>
  <param match="Designated Router: (.*)"
```

```
        replace="👑 Designated Router: $1">ospf.hello.designated_router</
param>
  <param>ospf.hello.network_mask</param>   <!-- no emoji — secondary -->
</ip-message>
```

Use full field names rather than abbreviations in the replacement to keep the output self-documenting.

7.9.2 Localized Hosts Files

The hosts file supports emoji and multilingual text in hostnames, so axis headers can carry the same bilingual treatment as message labels. Map both the IP and MAC entries of a device to the same display name so the label is consistent whether the message is carried over UDP or directly over Ethernet:

```
# PROFINET Hosts — English + Japanese with emoji
192.168.1.1       🏭 PLC-Siemens / PLCコントローラ
00:1c:06:0b:26:ed 🏭 PLC-Siemens / PLCコントローラ
192.168.1.2       🔧 VersaMax / バーサマックス
01:0e:cf:00:00:00 📣 PN-DCP-MC / DCPマルチキャスト
```

Mapping multiple addresses (IPv4 + IPv6, several per-port MACs of a switch, NAT'd peers) to the same display name merges them onto a single diagram axis — the same device no longer shows up as two columns.

VisualEther reads `.` (dot) in a hostname as a **child.parent** hierarchy separator (like DNS), so `Router1.AS65000.Network` renders as a three-level grouped label. Avoid stray dots in display names that aren't hierarchical (e.g., `PLC v2.0` would split on the version number); use a middle dot (`·`) or omit the version.

> **Tip**
>
> Use `visualether endpoints` (or the MCP `extract_endpoints` tool) as a checklist of every IPv4/IPv6 address in the capture, so nothing is missing from the hosts file.

7.9.3 Parallel FXT and Hosts Files per Language

Because localization lives entirely in `replace` strings and host names, you can maintain parallel files (`explore.fxt.xml` + `hosts.txt` , `explore-ja.fxt.xml` + `hosts-ja.txt` , `explore-ko.fxt.xml` + `hosts-ko.txt` , …) with identical `match` patterns and field references. Only the labels and host names differ.

7.9.4 Font Configuration for Non-Latin Scripts

To render CJK, Devanagari, Arabic, Hebrew, or other non-Latin scripts in PDF output, set `VISUALETHER_FONTS_DIR` to a directory containing the required `.ttf` / `.otf` font files. The fonts apply to both message labels and axis headers. See Section 14.1 for configuring this in the MCP server, or pass the variable when running the CLI.

> **Tip**
>
> When using Claude Code, ask it to generate translated FXT templates and hosts files together: *"Please create explore.fxt.xml and a hosts file from capture.pcap with Japanese translations and emoji."* Claude Code maps device roles to appropriate emoji, translates entity names, and covers both IP and MAC addresses.

7.10 Next Steps

Section 10 covers how to pair `style` attributes with the labels produced here so that protocol roles are visually distinguishable. Section 11 covers how to share opcode/param sets across templates with `<message-base>` and `extends`, so the patterns in this chapter need to be written only once.

8 Session Tracking

Sessions group related messages for lifecycle tracking. A session represents a sequence of related messages between network entities — for example, a TCP connection, a DNS query/response pair, a BGP peering session, or a 5G signaling flow.

8.1 Session Type Definition

A `<session-type>` element defines how sessions are identified and qualified:

```
<session-type name="bgp-peer" default="true">
  <!-- Optional: filter which packets qualify -->
  <qualify logic="or">
    <field value="179">tcp.srcport</field>
    <field value="179">tcp.dstport</field>
  </qualify>

  <!-- Required: define the session identity key -->
  <key>
    <source><field>ip.src</field></source>
    <destination><field>ip.dst</field></destination>
  </key>

  <!-- Optional: fallback key if primary fields are missing for some
       packets (e.g., NAT or capture-location asymmetry). -->
  <fallback-key>
    <source><field>ip.src</field></source>
    <destination><field>ip.dst</field></destination>
  </fallback-key>
</session-type>
```

8.1.1 Session Type Attributes

Attribute	Type	Default	Description
`name`	string	required	Unique identifier for this session type
`default`	boolean	`false`	Use when no `session-type` specified on a message
`direction-agnostic`	boolean	`true`	Treat A->B and B->A as the same session. Set `="false"` only for directional protocols (e.g. ARP request->reply, syslog client->server)

`max-duration`	number	none	Total session lifetime cap (seconds since first message)
`idle-timeout`	number	none	Inactivity gap cap (seconds since last message). Distinct from `max-duration` — see Section 9
`auto-start`	boolean	`false`	Create session on first qualifying packet if none exists

8.2 Session Qualification (`<qualify>`)

Qualification defines which packets are eligible for a session type. Without it, a session type could inadvertently match unrelated traffic.

8.2.1 Basic Port Filtering

```
<!-- Match packets where source OR destination port is 179 (BGP) -->
<session-type name="bgp-peer">
  <qualify logic="or">
    <field value="179">tcp.srcport</field>
    <field value="179">tcp.dstport</field>
  </qualify>
  <key>...</key>
</session-type>
```

8.2.2 Field Constraint Types

Attribute	Description	Example
`value`	Exact match or comma-separated list	`value="80,443,8080"`
`value-min` / `value-max`	Inclusive range (both required)	`value-min="1024" value-max="65535"`
`value-regex`	Regular expression pattern	`value-regex="^10\."`

Only one constraint type can be used per field.

8.2.3 Logic Combinations

Structure	Logic
Multiple `<qualify>` elements	**AND** between them

Fields within `<qualify logic="or">`	**OR** between fields
Fields within `<qualify>` (default)	**AND** between fields

Multiple `<qualify>` elements combine with an implicit AND:

```
<!-- (srcport=22 OR dstport=22) AND (ip.src is not private) -->
<session-type name="ssh-external">
  <qualify logic="or">
    <field value="22">tcp.srcport</field>
    <field value="22">tcp.dstport</field>
  </qualify>
  <qualify>
    <field value-regex="^(?!10\.|192\.168\.)">ip.src</field>
  </qualify>
  <key>...</key>
</session-type>
```

8.2.4 Not-Exists Condition

The `<not-exists>` element matches packets where a specific field is **absent**:

```
<!-- Match QUIC packets only when HTTP/3 is NOT present -->
<qualify logic="and">
  <field value="443">udp.dstport</field>
  <not-exists>http3.frame_type</not-exists>
</qualify>
```

This is useful for creating fallback logic that matches only when higher-layer protocols are absent.

8.2.5 Common Qualification Patterns

Protocol	Qualify Pattern
BGP	`<field value="179">tcp.srcport/dstport</field>`
DNS	`<field value="53">udp.srcport/dstport</field>`
HTTP	`<field value="80,443,8080,8443">tcp.srcport/dstport</field>`
SIP	`<field value="5060">udp.srcport/dstport</field>`
NGAP	`<field value="38412">sctp.srcport/dstport</field>`
S1AP	`<field value="36412">sctp.srcport/dstport</field>`

Diameter	`<field value="3868">sctp.srcport/dstport</field>`
GTP-U	`<field value="2152">udp.srcport/dstport</field>`
PFCP	`<field value="8805">udp.srcport/dstport</field>`

> **Important**
>
> The `<qualify>` element is **only** valid inside `<session-type>`. Placing it inside a message template (`<tcp-message>` and similar elements) causes a validation error. Use `match` on `<opcode>` to filter messages within a template.

8.3 Session Keys (`<key>`)

The `<key>` element defines how to identify unique sessions — effectively the “primary key” used to group related packets.

8.3.1 Option A: Source/Destination Pairs

Use when sessions are between two endpoints:

```
<key>
  <source>
    <field>ip.src</field>
    <field>tcp.srcport</field>
  </source>
  <destination>
    <field>ip.dst</field>
    <field>tcp.dstport</field>
  </destination>
</key>
```

`<source>` and `<destination>` must contain the same number of fields.

8.3.2 Option B: Protocol-Only Keys

Use when a single identifier uniquely identifies the session:

```
<key>
  <protocol><field>sip.Call-ID</field></protocol>
</key>
```

8.3.3 Option C: Combined Keys

Use when you need both endpoint information and protocol identifiers:

```
<key>
  <source><field>ip.src</field></source>
  <destination><field>ip.dst</field></destination>
  <protocol><field>dns.id</field></protocol>
</key>
```

8.3.4 Option D: IP-Version-Agnostic Keys (`addr.src` / `addr.dst`)

For dual-stack captures, use the synthetic `addr.src` / `addr.dst` fields instead of `ip.src` / `ip.dst` . The engine populates `addr.*` from `ip.*` on IPv4 packets and from `ipv6.*` on IPv6 packets, so a single key shape forms correctly on both address families:

```
<session-type name="bgp-session">
  <key>
    <source>
      <field>addr.src</field>
      <field>tcp.srcport</field>
    </source>
    <destination>
      <field>addr.dst</field>
      <field>tcp.dstport</field>
    </destination>
  </key>
</session-type>
```

Required under `auto-v6="true"` . The `auto-v6` expansion generates IPv6 message templates from their IPv4 siblings (see Section 6.1.1), but the session-type's key is shared between both transport variants. If the key uses IPv4-only fields, every IPv6 message matched by the auto-generated v6 template has no session to attach to and is silently dropped. The `check_v4_only_keying_under_auto_v6` load-time validator warns when this happens and suggests the `addr.*` rewrite verbatim.

Natural fit for `--merge-inputs` . `addr.*` resolves to the literal IP address and reads identically whether the run is single-pcap or merged.

When you have IPv4-only or IPv6-only protocols (NDP, ICMPv6, RIPng), prefer the explicit `ip.*` or `ipv6.*` form — the address-family distinction is intentional and `addr.*` would erase it.

8.3.5 Common Key Patterns

Protocol	Key Pattern
TCP	`tcp.stream` (Wireshark conversation index, default). Switch to the 4-tuple via `addr.src` / `addr.dst` + `tcp.srcport` / `tcp.dstport` (see Section 8.3.4) when participant axes must come straight from the key, or when composing with an app-layer ID.

UDP	`udp.stream` (Wireshark conversation index, default). Same composition trade-off as TCP.
SCTP	`sctp.assoc_index` when enabled; otherwise the 4-tuple. `sctp.assoc_index` is disabled by default in tshark, so most SCTP samples key on the 4-tuple in practice.
DNS	`dns.id`
BGP	`tcp.stream`
SIP	`sip.Call-ID`
NGAP	`ngap.AMF_UE_NGAP_ID` + `ngap.RAN_UE_NGAP_ID`
GTP-U	`gtp.teid` with IP pair

8.3.6 Cross-File Session Tracking (`--merge-inputs`)

When processing multiple PCAP files with `--merge-inputs` , VisualEther fronts dissection with `mergecap -w - | tshark -r -` : the input files are dissected as one continuous capture, so QUIC/TLS handshake state, HTTP/2 stream tables, TCP reassembly, and Wireshark stream indices (`tcp.stream` , `udp.stream` , `sctp.assoc_index` , `frame.number`) are all globally consistent across the merged stream. Wireshark's normal session-key conventions apply unchanged.

For plain TCP / UDP / SCTP session types, key on the Wireshark stream index — a single field replaces the four-field source/destination pair and matches what Wireshark groups under **Statistics → Conversations**:

```
<session-type name="ftp-control-session">
  <key>
    <protocol>
      <field>tcp.stream</field>
    </protocol>
  </key>
</session-type>
```

The choice between `tcp.stream` and an explicit 4-tuple is independent of merging — both stay globally consistent across the merged stream, so cross-file tracking works either way, and `tcp.stream` stays the simpler default. Reach for a 4-tuple key (via the IP-version-agnostic `addr.src` / `addr.dst` — see Section 8.3.4 — paired with the transport ports) only when the bare stream index can't carry what you need: when the sequence-diagram axes should be the IP endpoints named in the key, or when you fold the key together with an app-layer stream ID like `http2.streamid` so the multiplexed sub-sessions still read as endpoint-to-endpoint conversations.

A `<fallback-key>` is not needed for plain TCP/UDP/SCTP sessions — the primary key field is present in every packet, so the fallback never fires. Add a fallback only when the primary fields are genuinely absent for a class of packets (e.g., NAT or capture-location asymmetries).

> **Tip**
>
> `--merge-inputs` is **strict**: the listed `--input` files must be in chronological order, and the gap between consecutive files must not exceed 300 seconds. VisualEther runs a `capinfos` pre-pass before invoking mergecap and rejects the run if a later file starts before the previous one ended or if the gap is too large — merging misordered or unrelated captures would silently produce meaningless sessions. See the Troubleshooting chapter for the specific error messages and fixes.

> **Tip**
>
> The built-in `ftp` sample includes a `split/` folder with the same capture split into five sequential files, simulating CI/CD capture rotation (`dumpcap -b`). Use it to test cross-file session tracking:
>
> ```
> visualether generate --fxt samples/ftp/sessions.fxt.xml \
>   --input "samples/ftp/split/*.pcap" --merge-inputs \
>   --output out --hosts-file samples/ftp/hosts.txt
> ```
>
> Sessions are reconstructed correctly across file boundaries, producing the same result as the single `ftp.pcap` .

8.4 Fallback Keys

When primary key fields are missing for some packets because of NAT, capture location, or protocol offloading, a `<fallback-key>` provides an alternative key that fires only on packets where the primary fails.

A typical fallback widens or narrows the primary's field set rather than swapping to a different family. For example, a primary 4-tuple keyed on `ip.src + tcp.srcport + ip.dst + tcp.dstport` can fall back to just `ip.src + ip.dst` for packets where the TCP layer wasn't dissected:

```
<session-type name="tcp-4tuple">
  <key>
    <source>
      <field>ip.src</field>
      <field>tcp.srcport</field>
    </source>
    <destination>
      <field>ip.dst</field>
      <field>tcp.dstport</field>
    </destination>
  </key>
  <fallback-key>
    <source><field>ip.src</field></source>
    <destination><field>ip.dst</field></destination>
```

```
  </fallback-key>
</session-type>
```

Do not add a fallback “just in case” — on plain TCP/UDP/SCTP transports the primary `tcp.stream` / `udp.stream` / `sctp.assoc_index` is always present, so the fallback never fires and becomes dead config that future authors are likely to copy.

The fallback key does **not** need to match the structure of the primary key.

8.4.1 Fallback Chain (`fallback="true"`)

The `fallback="true"` attribute on a key group selects, **per packet**, the first listed field that has a non-empty value. It is a choice among alternative fields on a single packet — **not** a correlation across them. If the selected field’s value differs from one packet to the next, those packets form **separate** sessions; the chain does not stitch changing identifiers together.

The `http3` sample uses it to prefer Wireshark’s stable QUIC connection number, falling back to the UDP port only on packets where that number hasn’t been computed:

```
<session-type name="http3-connection">
  <key>
    <source><field>addr.src</field></source>
    <destination><field>addr.dst</field></destination>
    <protocol fallback="true">
      <field>quic.connection.number</field>  <!-- preferred when present -->
      <field>udp.srcport</field>             <!-- used only when it is absent
-->
    </protocol>
  </key>
</session-type>
```

Use it when a packet may carry any one of several interchangeable identifiers and not all are present at once — `quic.connection.number` else the UDP port above, or `ngap.RAN_UE_NGAP_ID` else `ngap.AMF_UE_NGAP_ID` . Note what this does **not** do: QUIC connection-ID rotation is absorbed by `quic.connection.number` itself (Wireshark follows the migration and emits one stable number per connection), not by the fallback chain — the chain only covers packets where the preferred field is missing, never values that change over the session’s life.

8.5 Multi-Value Key Fields (Multiplexed Protocols)

Fallback keys cover the case where one of several alternative fields might be present. **Multi-value key fields** are the complementary case: a single field is present once but carries several values, each one representing a different logical session. Common examples include HTTP/2 streams multiplexed over a single TCP connection, and HTTP/3 streams multiplexed over a single QUIC connection.

For these protocols, choose a session key that includes the stream identifier. VisualEther then automatically matches the packet to each relevant session, even when a single packet contains data for multiple streams. You do not need any special FXT syntax for this — you only need the right key fields.

> **Important**
>
> This is **not** the same mechanism as `multi-match="true"`. Multi-value key splitting (this section) takes a **single** template + session-type and creates several sessions because the key field has several values in one packet. `multi-match="true"` (see Section 5.2) takes **several templates** and lets each one feed a **different** session-type from the same packet. The two are independent and often combined.

> **Tip**
>
> Single-stream protocols are unaffected. If a packet contains only one value for the key field, VisualEther behaves exactly as you would expect and tracks a single session.

HTTP/2 FXT session key:

```
<session-type name="http2-stream">
  <key>
    <source><field>ip.src</field><field>tcp.srcport</field></source>
    <destination><field>ip.dst</field><field>tcp.dstport</field></destination>
    <protocol><field>http2.streamid</field></protocol>
  </key>
</session-type>
```

HTTP/2 example: If one TCP packet on a given 4-tuple contains frames for HTTP/2 streams 3 and 5, the packet is added to two separate sessions — one for stream 3 and one for stream 5. The 4-tuple reads naturally as the endpoints involved and matches Wireshark's own conversation view.

HTTP/3 FXT session key:

```
<session-type name="http3-stream">
  <key>
    <protocol>
      <field>quic.connection.number</field>
      <field>http3.frame_streamid</field>
    </protocol>
  </key>
</session-type>
```

HTTP/3 example: If one QUIC packet on connection 5 contains HTTP/3 streams 12 and 8, the packet is added to two separate sessions — one per stream.

8.6 Direction-Agnostic Sessions

VisualEther is direction-agnostic **by default**: when keys include source and destination, packets from A to B and B to A are grouped into the **same** session. This is what most request/response and bidirectional protocols want, so no attribute is needed in the common case:

```
<session-type name="tcp-4tuple">
  <key>
    <source>
      <field>ip.src</field>
      <field>tcp.srcport</field>
    </source>
    <destination>
      <field>ip.dst</field>
      <field>tcp.dstport</field>
    </destination>
  </key>
</session-type>
```

Set `direction-agnostic="false"` **only** when the two directions represent semantically distinct sessions — ARP request vs. reply, syslog client→ server, or any protocol where A→ B and B→ A should appear as separate sequence-diagram threads.

Use Case	Setting
IP/port tuple sessions (TCP, UDP, request/ response)	default (omit the attribute)
Protocol identifiers (Call-ID, stream index)	default (irrelevant — no source/destination to swap)
UE identifiers (NGAP, RRC)	default
ARP request→ reply	`direction-agnostic="false"`
Syslog or other strictly directional flows	`direction-agnostic="false"`

8.7 Mid-Capture Session Handling

For captures that begin mid-connection (no TCP SYN), enable `auto-start` to create sessions on the first qualifying packet:

```
<session-type name="modbus-session" auto-start="true">
  <qualify logic="or">
    <field value="502">tcp.srcport</field>
    <field value="502">tcp.dstport</field>
  </qualify>
```

```
  <key>
    <source><field>addr.src</field><field>tcp.srcport</field></source>
    <destination><field>addr.dst</field><field>tcp.dstport</field></
destination>
  </key>
</session-type>
```

Scenario	auto-start=false (default)	auto-start=true
First packet is `session-start`	Session created	Session created
First packet is regular message	Discarded	Session auto-created
First packet is `session-stop`	Discarded	Discarded
Outcome (no explicit close)	`incomplete::no-close`	`incomplete::mid-capture`

> **Tip**
>
> Use `auto-start` for industrial protocols (Modbus, S7comm) with long-lived connections, rolling capture buffers, or any capture that starts after connections were established.

8.8 Parallel Session Types

VisualEther can derive **several** sessions from one packet stream by declaring multiple `<session-type>` definitions that the same frames feed at once. This always requires `multi-match="true"` on the root `<FXT>` (see Section 5.2). It comes in two flavors — the difference is whether you slice the traffic along the protocol stack or along time:

- **Per-layer views** (vertical slices): one session per protocol layer — MAC, RLC, PDCP, RRC, NAS, plus SDAP, IP, and SIP once the user plane is decrypted — all from the same frames. The 5G NR Radio showcase walks these through end to end (Section 4; the frame-level mechanic is in Section 5.2.1).
- **Phase sub-sessions** (horizontal slices): one session per internal phase of a single connection's lifetime — PAP authentication, IPCP negotiation, and the long-lived link, all within PPP. The rest of this section covers this flavor.

A long-lived session (a TCP connection, a PPPoE link, a 5G PDU session) typically passes through several internal phases — transport setup, authentication, parameter negotiation, steady-state data — before tearing down. If only the long-lived session is tracked, a capture that ends before teardown reads as `incomplete`, and the fact that authentication or address negotiation actually succeeded is invisible in the outcome summary.

The fix is to track **parallel** session types on the same flow: one long-lived session covering the whole connection, plus one phase-level session per internal milestone. Each phase session opens when the phase starts and closes with a clear `success::*` or `failure::*` outcome, regardless of whether the outer connection ever tears down.

8.8.1 When to Use Phase Sub-Sessions

Use phase sub-sessions when:

- A connection has internally meaningful phases (auth, key exchange, address assignment, capability negotiation) whose pass/fail status matters independently of overall teardown.
- Captures routinely end mid-flow, so the long-lived session reads `incomplete` even when the bring-up fully succeeded.
- The protocol uses different correlator fields per phase (e.g., PPP uses `pap.identifier` for PAP, `chap.identifier` for CHAP, `ppp.identifier` per IPCP round), and you want round-by-round outcome granularity.

8.8.2 Required: `multi-match="true"`

Each message template carries a single `session-type` attribute, so a packet can only feed multiple session types when `multi-match="true"` is set on the root `<FXT>` element. Without it, first-match-wins picks one template, and the other session types never see the packet.

```
<FXT multi-match="true">
  ...
</FXT>
```

See Section 5.2 for the full semantics. The relevant guarantee here is that **every** matching template fires, and each updates its own session type independently.

The inverse of this pattern is **partitioning** rather than sharing. When several same-shaped session types should each receive a **disjoint** slice of the frames — for example RDP carried on different transports, all running the same stack: `direct-rdp` on 3389, `vmconnect` on 2179, `rd-gateway` on 443, `kdc` on 88 — you do **not** use multi-match. The opcode fields are identical across transports, so instead gate each session type's templates with a `<param required="true" match="…">tcp.port</param>` and a frame falls through to the template (and session type) for its own transport (Section 7.6.1). One transport can stay ungated as the fall-through default.

8.8.3 Pattern Structure

- **One long-lived session** keyed by the connection-wide identifier — typically the transport 4-tuple, optionally folded with a protocol-specific ID like `pppoe.session_id` or `ngap.RAN_UE_NGAP_ID`. Stop it only on explicit teardown frames.
- **One phase session per internal milestone** keyed by the phase's natural correlator, with a `<qualify>` constraint requiring a phase-specific field so sibling phases that share the correlator don't feed the wrong session.
- **Parallel message templates** — the long-lived session keeps its display templates; each phase session adds its own templates with `session-start` / `session-stop` / `session-result`.
- `auto-start="true"` on the phase session when its opening frame is a regular message rather than an explicit start (typical for negotiation rounds that begin with a Configuration-Request).

8.8.4 Worked Example: PPPoE

The bundled `pppoe` sample tracks five session types in parallel on one PPPoE connection:

Session Type	Key	Stops On
`pppoe-connection`	`pppoe.session_id`	PADT, LCP Terminate-Ack, PAP Auth-Nak, CHAP Failure
`pap-auth`	`pap.identifier`	PAP Auth-Ack (success), Auth-Nak (failure)
`chap-auth`	`chap.identifier`	CHAP Success, Failure
`ipcp-negotiation`	`ppp.identifier` (qualified by `ipcp.opt.type`)	Configuration-Ack/Nak/Reject
`ipv6cp-negotiation`	`ppp.identifier` (qualified by `ipv6cp.opt.type`)	Configuration-Ack/Nak/Reject

The phase session types share `ppp.identifier`, so each one qualifies on a phase-specific field's presence to keep sibling phases from feeding the wrong session:

```
<session-type name="ipcp-negotiation" auto-start="true">
  <qualify>
    <field value-regex=".+">ipcp.opt.type</field>
  </qualify>
  <key><protocol><field>ppp.identifier</field></protocol></key>
</session-type>
```

`value-regex=".+"` matches any non-empty value, so only IPCP frames feed `ipcp-negotiation` — an LCP Configuration-Ack with the same `ppp.identifier` no longer closes an in-flight IPCP round by accident.

For a capture that establishes a PPPoE link and ends without teardown, the parallel session types produce:

```
pppoe-connection/incomplete:    1   (no PADT — accurate)
pap-auth/success:               1   (authentication succeeded)
ipcp-negotiation/success:       2   (two IPCP rounds Acked)
ipcp-negotiation/failure:       1   (one Configuration-Nak round)
ipv6cp-negotiation/incomplete:  2   (BRAS used LCP Protocol-Reject)
```

Instead of a single `pppoe-connection/incomplete` line, the outcome summary now directly answers "Did auth succeed?", "How many IPCP rounds were needed?", and "What happened to IPv6CP?" without opening the diagram.

> **Tip**

> The same pattern applies to other layered flows: track a TCP session alongside per-request HTTP sessions, an SCTP association alongside per-procedure NGAP sessions, or a QUIC connection alongside per-stream HTTP/3 sessions. The long-lived session captures the connection lifetime; the phase sessions capture the bring-up and per-transaction outcomes.

8.9 Next Steps

This chapter covered the **shape** of sessions — types, keys, qualifiers, and parallel sub-sessions. Section 9 covers the **lifetime** of each session: start triggers, stop conditions, timeouts, and outcome classification.

9 Session Lifecycle

Session lifecycle management defines how sessions begin, progress, and end. This chapter covers lifecycle attributes, result classification, session timeouts, and CI/CD filtering.

9.1 Lifecycle Attributes

Session boundaries are controlled by attributes on message templates:

Attribute	Description
`session-start="true"`	First message creates a new session
`session-stop="true"`	Message terminates the session
`session-result="..."`	Outcome classification (only with `session-stop`)

> **Important**
>
> A message cannot be both `session-start` and `session-stop`. Regular messages, without either attribute, are automatically associated with an existing active session.

9.2 Lifecycle Example

```
<!-- TCP SYN: Session Start -->
<tcp-message session-type="bgp-session" session-start="true"
             style="slate dashed">
  <opcode match=".*\((SYN)\)"
          replace="TCP ($1)">tcp.flags</opcode>
</tcp-message>

<!-- BGP OPEN: Mid-session message -->
<tcp-message session-type="bgp-session" style="royal-blue">
  <opcode match="^Type: (OPEN) Message \(.*\)"
          replace="BGP $1">bgp.type</opcode>
  <param>bgp.open.myas</param>
  <param>bgp.open.holdtime</param>
</tcp-message>

<!-- TCP FIN: Graceful session end -->
<tcp-message session-type="bgp-session" session-stop="true"
             session-result="success::graceful-fin"
             style="forest-green dashed">
```

```
  <opcode match=".*\((.*FIN.*)\)"
          replace="TCP ($1)">tcp.flags</opcode>
</tcp-message>

<!-- TCP RST: Abnormal session end -->
<tcp-message session-type="bgp-session" session-stop="true"
             session-result="failure::tcp-reset"
             style="brick wave">
  <opcode match=".*\((.*RST.*)\)"
          replace="TCP ($1)">tcp.flags</opcode>
</tcp-message>
```

9.3 Session Result Format

The `session-result` attribute uses the format:

```
<category>::<reason>
```

The category is mandatory. The reason is optional and provides additional detail.

9.3.1 Outcome Categories

Category	Meaning
`success`	Session completed normally
`failure`	Session terminated due to an error
`incomplete`	Session did not complete as intended
`late`	Session exceeded `max-duration` threshold (response received after deadline)
`timeout`	No response received within `max-duration`

9.3.2 Example Results

The category controls filtering and output grouping. The reason is a short descriptive label chosen by the template author.

Example	Meaning
`success::graceful-fin`	TCP session closed normally with FIN
`failure::tcp-reset`	TCP session ended with RST
`incomplete::no-close`	Session never reached a normal close condition

`late::exceeded max duration of 30.000s`	Response arrived, but after the `max-duration` deadline
`timeout::exceeded max duration of 30.000s`	No response arrived before the `max-duration` deadline

9.3.3 Protocol-Specific Results

Protocol	Common Results
TCP	`success::graceful-fin`, `failure::tcp-reset`, `incomplete::syn-no-ack`
BGP	`success::tcp-close`, `failure::bgp-notification`, `failure::hold-timer-expired`
DNS	`success::dns-response`, `failure::server-failure`, `incomplete::no-response`
SIP/RTP	`success::call-completed`, `failure::call-rejected`, `incomplete::rtp-timeout`
5G NGAP	`success::normal-release`, `failure::abnormal-ue-release`, `failure::radio-failure`

9.3.4 Naming Guidelines

- Use **kebab-case** (`abnormal-ue-release`)
- Keep reasons **short and stable**
- Never overload the category — use the reason for detail

9.4 Session Timeout

Two independent attributes can bound a session's lifetime — pick the one that matches the protocol's notion of "too long":

Attribute	What it measures	Outcome when exceeded
`max-duration`	Total wall-clock since the session's **first** message	`late::…` (stop arrives after deadline) or `timeout::exceeded max duration of Xs`
`idle-timeout`	Gap since the session's **last** message (inactivity)	`timeout::exceeded idle timeout of Xs`

9.4.1 `max-duration` — absolute lifetime cap

```
<session-type name="api-call" max-duration="30">
  <key>
    <source><field>ip.src</field><field>tcp.srcport</field></source>
    <destination><field>ip.dst</field><field>tcp.dstport</field></destination>
  </key>
</session-type>
```

When a session exceeds its `max-duration`, one of two outcomes applies:

- **Late** — A `session-stop` message arrives after the `max-duration` deadline. The session result is set to `late::exceeded max duration of Xs`, and the session is placed in the `late/` output directory.
- **Timeout** — No `session-stop` message arrives, and the `max-duration` is exceeded. The session result is set to `timeout::exceeded max duration of Xs`, and the session is placed in the `timeout/` output directory.

> **Tip**
>
> Use `max-duration` in CI/CD pipelines to flag sessions that take too long, detect hung connections, or enforce SLA compliance.

9.4.2 `idle-timeout` — inactivity timer

Use `idle-timeout` for protocols where a session can legitimately last hours but should be considered torn down after a configured period of silence — for example, the 3GPP RRC Inactivity Timer (TS 38.331), TCP keepalive, NAT mapping eviction, or SCTP heartbeat loss.

```
<session-type name="rrc-connection" idle-timeout="30" auto-start="true">
  <key>
    <protocol><field>mac-nr.rnti</field></protocol>
  </key>
</session-type>
```

If no message is observed on the session's key for more than the configured number of seconds, the session is closed with `timeout::exceeded idle timeout of Xs` (where X is the configured limit) and placed in the `timeout/` output directory. All idle timeouts for a given `session-type` share the same reason string, so CI/CD filters and the MCP `extract_sessions` summary bucket them together. A subsequent message on the same key opens a fresh session (when `auto-start="true"` or an explicit `session-start` trigger fires), modeling the protocol's "session ended, new one begins" semantics.

The check fires both **mid-capture** (when a new packet arrives after a long silence) and **at the end of capture** (when the trailing idle gap to the capture's last timestamp exceeds the limit). The latter replaces what would otherwise be `incomplete::mid-capture` on captures that genuinely went idle before the recording stopped.

`idle-timeout` and `max-duration` may be set on the same `<session-type>`. They are independent: `max-duration` is checked first, so an absolute lifetime cap takes precedence over an inactivity gap when both fire.

9.5 CI/CD Filtering

Filter sessions by outcome to focus on failures in automated pipelines:

```
<session-filter outcome-in="failure incomplete" />
```

outcome-in Value	Sessions Included
`success`	Successfully completed
`failure`	Error conditions
`late`	Response received after max-duration deadline
`timeout`	No response within max-duration
`incomplete`	No proper closure

Multiple values are space-separated. If `<session-filter>` is omitted, all sessions are included.

Use Case	Recommended Filter
Troubleshooting	`success failure incomplete`
Automated testing	`failure incomplete`
Strict compliance	`failure`

> **Warning**
>
> CI/CD filtering operates **only** on the category, the part before `::`, not the reason. When no `<session-filter>` is specified in the FXT, all outcomes are extracted. Use `--session-filter` at the command line to override the FXT setting at runtime.

9.6 Output Structure

Once VisualEther has classified every session by `session-result`, it writes them to disk in a directory layout that mirrors that classification. Sessions are organized under a `<pcap-stem>/` subdirectory, named after the input capture file, and then by type and result:

```
<output_dir>/
  <pcap-stem>/
```

```
    <session-type>/
      <session-result>/
        00001_viewer.html
        00002_viewer.html
        ...
```

This structure is consistent across all modes — single-file, independent multi-file, and merge mode each produce a `<pcap-stem>/` subdirectory inside the output directory.

For example, a BGP session that ends with a TCP reset would be placed in:

```
output/bgp/bgp-session/failure/00001_viewer.html
```

9.7 Next Steps

Section 12 covers how the per-session diagrams under each `<session-result>/` are rendered (combined viewer, lazy mode, NDJSON sidecars) and how the Session Navigator surfaces them.

10 Styling

VisualEther supports visual styling of message arrows in sequence diagrams through the `style` attribute on message elements. Styling conveys semantic meaning through color and line style.

10.1 Style Attribute Format

```
style="<color> <arrow-style>"
```

- Components can appear in **any order**.
- Both components are **optional**.
- Values are **case-insensitive**.

```
<!-- Just a color (solid arrow) -->
<tcp-message style="royal-blue">

<!-- Just an arrow style (default color) -->
<tcp-message style="dashed">

<!-- Color and arrow style -->
<tcp-message style="crimson wave">

<!-- Arrow style first, then color -->
<tcp-message style="dotted teal">

<!-- RGB color with arrow style -->
<tcp-message style="rgb(255, 100, 50) dashed">
```

10.2 Named Colors

VisualEther provides 30 named colors organized by category:

Category	Colors
Blues	`steel-blue`, `royal-blue`, `cerulean`, `midnight`, `indigo`
Greens	`sea-green`, `forest-green`, `emerald`, `pine`
Teals/Cyans	`cyan`, `turquoise`, `teal`
Oranges/Yellows	`tangerine`, `goldenrod`, `sand`
Purples	`amethyst`, `slate-blue`, `plum`, `orchid`

Reds/Pinks	`crimson` , `brick` , `magenta` , `maroon`
Browns	`saddle` , `sienna` , `rust` , `chocolate`
Neutrals	`slate` , `charcoal` , `graphite`

You can also specify custom colors using RGB format: `rgb(255, 128, 0)`

10.3 Arrow Styles

Style	Visual	Typical Use
`regular`	Solid line	Primary request/action messages (default)
`dashed`	Dashed line	Response/acknowledgment messages
`dotted`	Dotted line	Background/keepalive/heartbeat messages
`dot-dash` (or `dotdash`)	Dot-dash alternating	Cancel/terminate messages
`wave`	Wavy line	Error/failure messages

10.4 Monochrome Mode

By default, VisualEther renders PDFs in full color — message arrows use the colors specified by `style` attributes, and axis headers are colored automatically from a 30-color palette.

Use `--monochrome` to produce a completely black-and-white PDF suitable for printing or grayscale reproduction:

```
visualether generate --fxt my.fxt.xml --input capture.pcap \
  --output out --monochrome
```

In monochrome mode, all colors are removed from the entire PDF — axes, message arrows, and parameters are rendered in black. Arrow styles (`dashed` , `dotted` , `wave` , `dot-dash`) remain distinguishable, making them especially important for conveying semantic meaning in black-and-white output.

10.5 Next Steps

Styling controls how each message is rendered. To reduce repetition across many similar templates, see Section 11 for template groups, message inheritance, and priority overrides. To control how the styled diagrams are packaged (PDF, HTML, NDJSON, lazy mode), see Section 12.

11 Advanced FXT Features

This chapter covers advanced features for reducing repetition and reusing template definitions across FXT files.

11.1 Template Groups

Template groups let multiple message templates share common attributes, reducing repetition:

```
<template-group session-type="http-session" session-start="true"
                style="coral">
  <tcp-message>
    <opcode match="GET">http.request.method</opcode>
  </tcp-message>
  <tcp-message>
    <opcode match="POST">http.request.method</opcode>
  </tcp-message>
</template-group>
```

11.1.1 Shareable Attributes

The following attributes can be set on the `<template-group>` element:

- `style`
- `session-type`
- `bookmark`
- `priority`
- `session-start`
- `session-stop`
- `session-result`

Child message elements inherit all group attributes. If a child explicitly specifies an attribute, it **overrides** the group attribute. Groups are flattened during parsing and cannot be nested.

11.2 Template Inheritance (`message-base` and `extends`)

Template inheritance defines reusable parameter sets that can be shared across multiple templates.

11.2.1 Defining a Base

Use `<message-base>` to define a reusable set of parameters and an optional remark:

```
<message-base name="quic-transport" style="slate">
  <param>quic.dcid</param>
  <param>quic.packet_number</param>
```

```
  <remark>frame.time_relative</remark>
</message-base>
```

11.2.2 Using a Base

Reference the base with the `extends` attribute on a message template:

```
<udp-message extends="quic-transport">
  <opcode>http3.frame_type</opcode>
  <param>http3.headers.status</param>
</udp-message>
```

The `extends` attribute can be used on any message element (`<tcp-message>` , `<udp-message>` , and so on) to inherit parameters and a remark from a named `<message-base>` .

11.2.3 Inheritance Behavior

1. Base params are **prepended** to the template's own params.
2. A style is inherited only if the template does not specify one.
3. A remark is inherited only if the template does not specify one.
4. Multiple templates can extend the same base.
5. Bases can be defined anywhere in the file — forward references are allowed; only the **name** needs to exist.

11.2.4 When to Use Groups vs. Inheritance

Feature	Template Groups	Template Inheritance
Shares	Attributes (style, session-type, etc.)	Parameters and remark
Use case	Same session or style across templates	Same field extractions across templates

The two features are complementary: a template inside a `<template-group>` can also use `extends="…"` to pull in a base's params. Use groups to share **attributes** (one element wraps the children) and inheritance to share **fields** (each template names a base by reference).

11.3 Dual-Stack Twin Coverage

`auto-v6="true"` (see Section 6.1.1) and the load-time `FXT-TWIN` coverage check pair each transport template with its other-family twin by matching on `(opcode field, @match)` , then report any IPv4 template whose IPv6 counterpart (or vice versa) is missing. Two refinements keep the check from flagging templates that are correct as written — protocols whose v4 and v6 forms use entirely different field names, and protocols that exist on only one IP version.

11.3.1 Sibling-Protocol Pairs

The dual-stack twin coverage check (`FXT-TWIN`) recognises four IPv4↔IPv6 sibling-protocol pairs whose Wireshark field prefixes differ entirely (so a `(field, @match)` pairing would never match): `dhcp` ↔ `dhcpv6` (RFC 2131 / 8415), `rip` ↔ `ripng` (RFC 2453 / 2080), `icmp` ↔ `icmpv6` (RFC 792 / 4443), and `igmp` ↔ `mld` (RFC 3376 / 3810). A file that defines templates on both halves of a pair gets full coverage credit. Protocols whose field prefix is the same on both v4 and v6 (DNS, NTP, mDNS) are **not** in this map; they still need true `(field, @match)` symmetry, which `auto-v6="true"` provides automatically.

11.3.2 Single-Stack Protocols (`single-stack="true"`)

Some protocols are inherently single-stack and have no equivalent on the other IP version — NetBIOS Name Service (replaced on IPv6 by LLMNR and mDNS, not by NBNS-over-v6), NetFlow v5 (superseded by v9 and IPFIX for IPv6 coverage). Mark the template with `single-stack="true"` to opt out of both the `FXT-TWIN` check and `auto-v6` expansion:

```
<udp-message style="royal-blue" single-stack="true">
  <opcode match="Version: (5)" replace="📊 NetFlow v$1">cflow.version</
opcode>
</udp-message>
```

If `auto-v6="true"` is set on the root and a template carries `single-stack="true"` , no synthetic v6 twin is generated for that template.

11.3.3 Session Keys Under `auto-v6`

`auto-v6` only generates the message templates — the session-type's key is shared between IPv4 and IPv6 traffic. A key built from `ip.src` / `ip.dst` cannot form on the IPv6 packets matched by the auto-generated twin, so they are silently dropped. Use the synthetic `addr.src` / `addr.dst` fields (see "IP-Version-Agnostic Keys" in Section 8) so a single key shape works on both address families. The `check_v4_only_keying_under_auto_v6` validator warns at load time when a session-type referenced by an auto-v6 expansion still uses v4-only keying.

11.4 Next Steps

With the FXT format fully covered, Section 12 explains how to package the resulting diagrams (combined viewer, lazy mode, NDJSON, markdown). Section 13 applies everything so far to a protocol that doesn't ship with a sample.

12 Output Formats

VisualEther produces sequence diagrams in multiple formats, each suited to different workflows.

12.1 Available Formats

Format	Edition	Description
`auto`	All	Default. Resolves to `combined`, `lazy`, or `pdf` based on capture size and mode — see Auto-Downgrade below. On Community Edition, `auto` always resolves to `pdf`.
`combined`	Pro/ Server	Full PDF + HTML split-pane viewer with clickable message details. No automatic downgrade.
`pdf`	All	Standalone PDF sequence diagram with optional bookmarks.
`html`	Pro/ Server	HTML Flow Dossier — self-contained HTML with the sequence diagram pinned on the left and message details on the right. See Section 12.3.
`lazy`	Pro/ Server	NDJSON with a session navigator. PDFs render on demand via the auto-spawned `visualether serve`.
`ndjson`	Pro/ Server	Newline-delimited JSON for machine consumption and LLM analysis.
`markdown` / `md`	Pro/ Server	Markdown representation of the message flow.

Community Edition is restricted to `pdf` output — all other formats require a Professional or Server license. Multiple formats can be requested in a single run with repeated `--format` flags — for example, `--format pdf --format ndjson` to get both a PDF and the NDJSON sidecar.

12.2 Combined Viewer

The `combined` format produces an HTML viewer that embeds the PDF on the left and HTML message details on the right (Figure 6). Clicking a message in the PDF navigates to its details in the HTML pane.

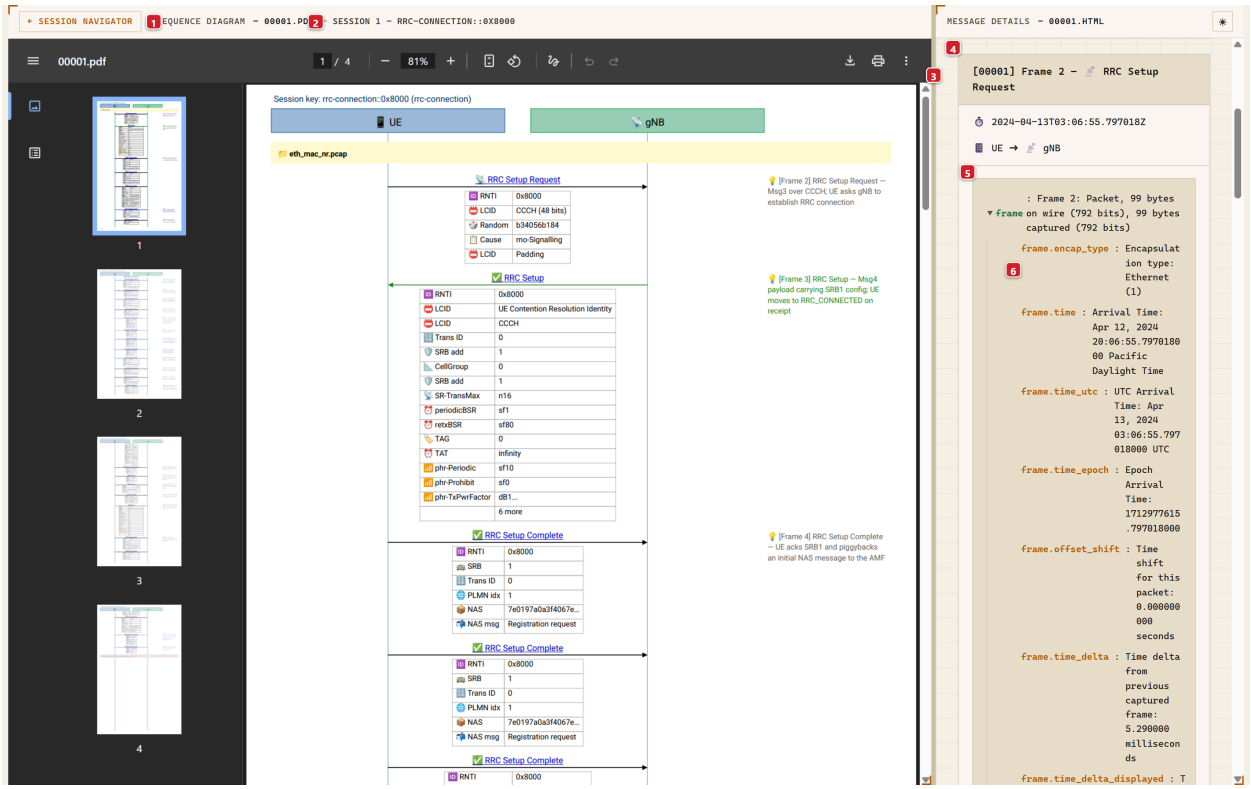

Figure 6: Combined Viewer — PDF on the left, expandable packet tree on the right.

1. Back to Session Navigator
2. Session title (type + key)
3. Draggable divider — resize the panes
4. Message card — frame number, timestamp, source, destination, opcode
5. Expandable packet tree (full Wireshark dissection)
6. Click any tree key or value to copy it

12.3 HTML Flow Dossier

The `html` format produces a self-contained HTML dossier for each flow: the sequence diagram pinned on the left, full message details flowing on the right (Figure 7). With a session template you get one dossier per session; with an explore template you get one per capture. Either way, it is the most direct way to walk the flow end to end — the diagram and its axes stay in view while you read each message, and clicking a label or arrow scrolls the matching details card into focus.

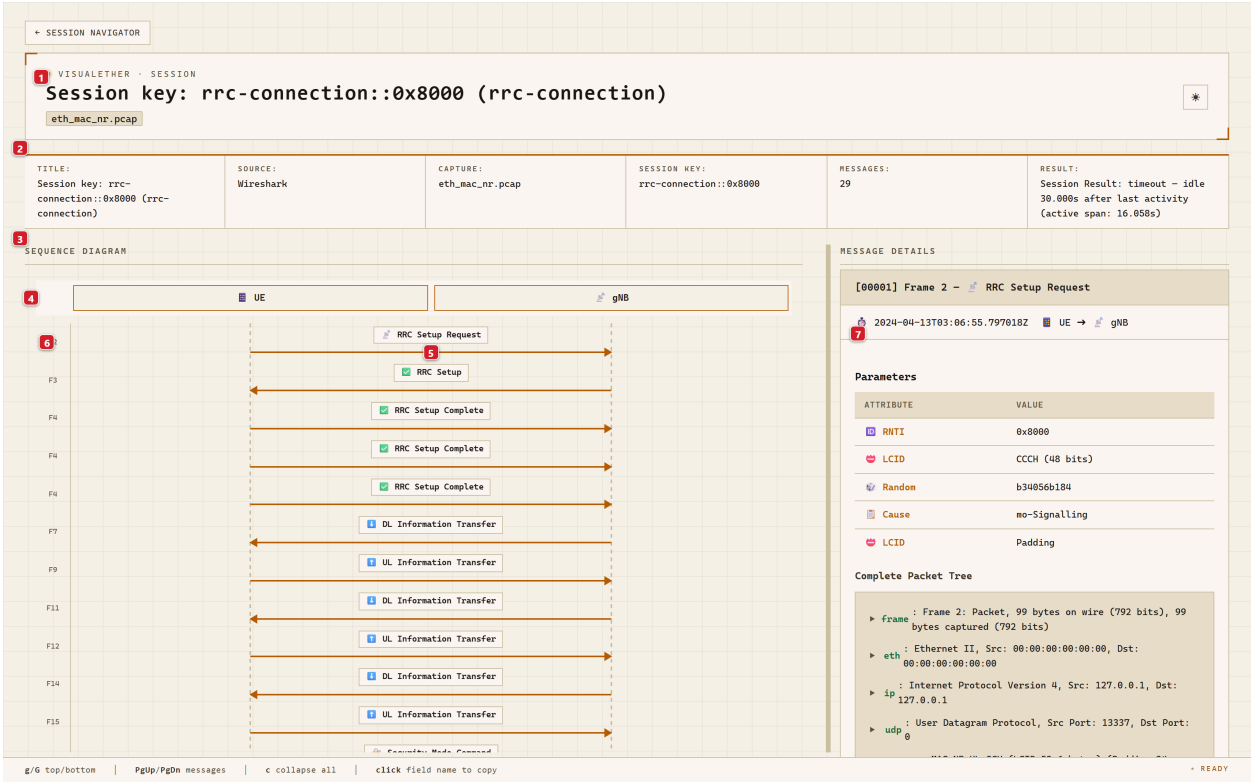

Figure 7: HTML Flow Dossier — sequence diagram on the left, message details on the right.

1. Masthead — session or capture title
2. Metadata strip — capture file, message count, result
3. Sequence diagram — pinned to the left pane during scroll
4. Axis headers — stay visible while scrolling long flows
5. Message arrow — click to jump to the matching details card
6. Frame-number gutter — left-aligned for vertical reading
7. Message details — parameters and the full packet tree per frame

The dossier works best when you want to read a flow end to end:

- **Small to medium flows** (up to a few hundred messages) where every step matters.
- **Sharing one flow with a teammate** — a self-contained `.html` file that opens directly in any browser.
- **Triage by colour** — in sessions mode the arrows pick up the result tint, so a failure stands out at a glance.

Generate a dossier with either template style:

```
visualether generate --fxt sessions.fxt.xml --input capture.pcap \
  --output out --format html      # one dossier per session
visualether generate --fxt explore.fxt.xml --input capture.pcap \
  --output out --format html      # one dossier per capture
```

For large dossiers, add `--gzip`. The dossier compresses to roughly a fifth of its uncompressed size and opens identically in the browser when served by `visualether serve`:

```
visualether generate --fxt explore.fxt.xml --input capture.pcap \
  --output out --format html --gzip
```

For very large captures, prefer `--format lazy`, which keeps the same dossier layout but renders each session on demand from the Session Navigator.

12.4 Auto-Downgrade for Large Captures

When `--format auto` (the default) is used, VisualEther selects the best format per input file, based on size and mode:

Condition	Mode	Selected format
≤ 15 MB	either	`combined` (interactive viewer)
> 15 MB	sessions	`lazy` (per-session PDFs rendered on demand)
> 15 MB	explore	`pdf` (single PDF; no per-session deferral makes sense)
> 1000 sessions	sessions	`lazy` (overrides `combined`, even below 15 MB)

12.4.1 Per-file resolution in batch mode

With multiple inputs (a glob, or several `--input` files) and **without** `--merge-inputs`, each capture is rendered as its own independent diagram, so `auto` resolves **per file** from that file's own size — not the batch total. A single `visualether generate --input *.pcap --input *.pcapng` run therefore renders the small captures as `combined` and a large one as `pdf` / `lazy` in the same pass. With `--merge-inputs` the inputs are one logical capture, so their combined (summed) size drives a single decision.

12.4.2 Session-count downgrade (sessions mode)

`combined` renders one PDF **eagerly per session**, so its cost scales with session count, not bytes. A small but session-dense capture can therefore cost far more than a larger capture with few sessions — a 3.5 MB SIPP trace with 2668 sessions renders combined in roughly 20 minutes, while an 11 MB http3 trace with 662 sessions renders in about 3. Byte size alone is blind to this, so in sessions mode `auto` also downgrades `combined` to `lazy` once a capture exceeds **1000 sessions** (`COMBINED_FORMAT_SESSION_CAP`), deferring per-session PDFs to on-demand rendering. VisualEther prints the choice, e.g.
`Format: lazy (2668 sessions > 1000 cap --- deferring per-session PDFs to on-demand rendering)`.

12.4.3 gzip inputs

For gzip-compressed inputs (`.pcap.gz`, `.pcapng.gz`), the size threshold uses the uncompressed size (read from the gzip ISIZE footer), so it reflects what `tshark` actually processes rather than the smaller

on-disk byte count. When `auto` selects a format by size, VisualEther prints the choice on startup, e.g., `Format: lazy (sessions mode, input 38 MB > 15 MB threshold)`.

To keep the combined viewer regardless of size or session count, pass `--format combined` explicitly — `auto`'s downgrades never override a format you request by name.

12.5 Lazy Format

> **Tip**
>
> Professional / Server only — `lazy` (and `auto` resolving to `lazy`) requires a Professional or Server license.

For large captures with many sessions, `lazy` lets you start exploring immediately — VisualEther writes the session navigator and per-session NDJSON (see NDJSON Output below) up front, then renders each PDF only when you actually click that session. The result is a much faster initial run and a much smaller output directory than rendering every session PDF eagerly.

Lazy mode honors any `--timestamp-format`. It decouples storage from display: the per-session NDJSON sidecars always store **absolute** timestamps (lossless), while your chosen display format is recorded in the manifest and applied per session when `serve` renders each PDF on demand. So `relative-first` and `relative-previous` (delta) work with lazy too — the deltas are computed against each session's own message sequence at render time. Add `--gzip` to compress the on-disk sidecars.

```
visualether generate --fxt sessions.fxt.xml --input large.pcap \
  --output out --format lazy --gzip
```

`auto` picks `lazy` only in sessions mode. You can still pass `--format lazy` explicitly with an explore template; VisualEther treats the entire capture as a single synthetic session, so the navigator has an entry point.

12.6 Gzip Compression

> **Tip**
>
> Professional / Server only — `--gzip` requires a Professional or Server license, since gzipped output is meant to be served over HTTP via `visualether serve`.

With `--gzip`, HTML and JSON files are written with a `.gz` extension. `visualether serve` transparently serves them with `Content-Encoding: gzip`. For static hosting:

- **nginx**: add `gzip_static on;` to the server block.
- **Caddy**: add `precompressed gzip` to the file server directive.

Clients never reference `.gz` in URLs — the server automatically handles content negotiation. Output size typically drops by roughly half.

12.7 Browser Compatibility

There are two ways to open the Combined Viewer in a browser, and they don't all work the same way:

- **Local file** (a `file://` URL) — you double-click `…_viewer.html` in Explorer / Finder, or drag it onto a browser window.
- **Web server** (an `http://` URL) — a server hosts the file, and the browser fetches it over HTTP. The Professional / Server auto-serve (`new` , `init` , or `generate --serve`) does this for you with `visualether serve` running in the background.

The combined PDF + HTML viewer with click-to-jump from the PDF works in these combinations:

Browser	Local file	Web server
Chrome / Edge	✓ works	✓ works
Firefox	Not supported	✓ works
Safari (macOS)	Not supported	Not supported

> **Warning**
>
> **Safari on macOS is not supported** for the Combined Viewer. Even when served over HTTP, clicks in the embedded PDF do not jump to the matching HTML details. Use Chrome, Edge, or Firefox on macOS instead — all three render the viewer fully when the output is opened over HTTP.

12.8 Session Navigator

When a `sessions.fxt.xml` template is used, VisualEther produces per-session sequence diagrams accessed from the Session Navigator (`index.html`). It provides search (press `/` to focus), result-category filtering, an interactive timeline with histogram buckets (Figure 8), and a tree view (Figure 9).

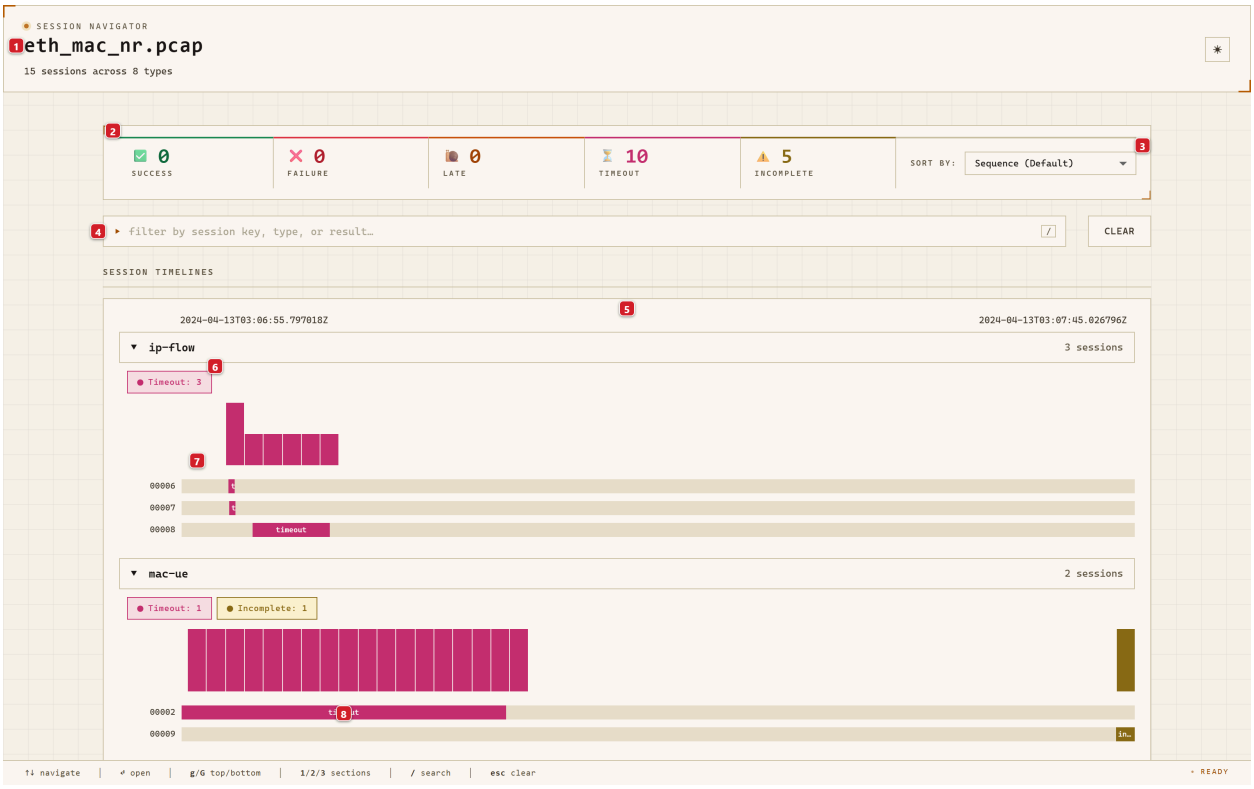

Figure 8: Session Navigator — timeline view with result chips, search, and per-session-type histograms.

1. Title — pcap file name
2. Result summary chips (Success / Failure / Late / Timeout / Incomplete)
3. Sort dropdown (Sequence / Timestamp / Messages / Duration)
4. Search bar — press `/` to focus; filters tree, timeline, and chips
5. Timeline header — absolute start and end timestamps
6. Result filter chips — click to filter by outcome
7. Histogram bucket — click to filter to sessions active in that interval
8. Timeline bar — click to open the session viewer

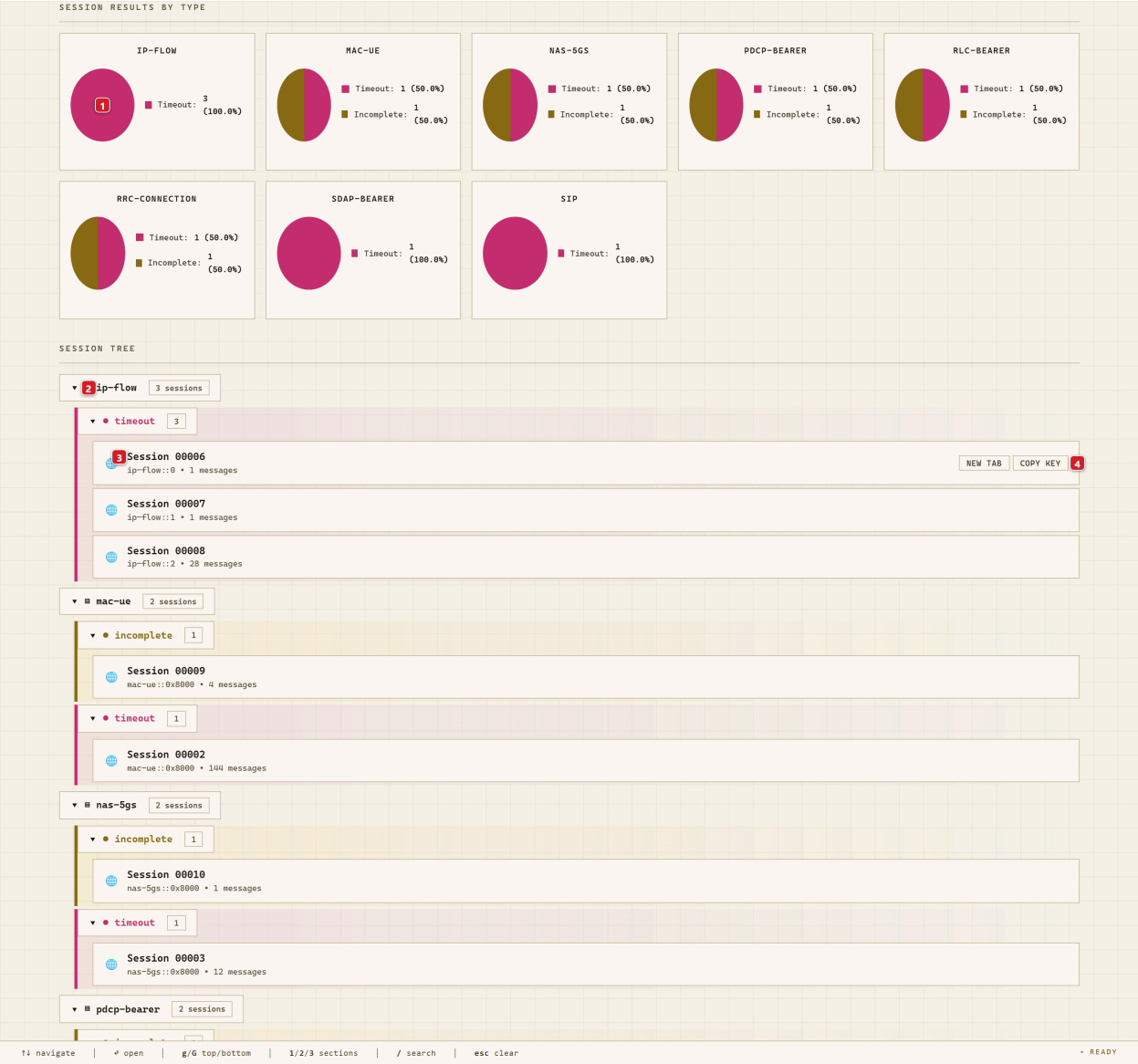

Figure 9: Session Navigator — tree view with per-type pie charts and session links.

1. Pie chart per session type, segmented by result
2. Tree view — Session Type → Result → Sessions
3. Session link — sequence number, key, message count, duration
4. Hover for quick actions — copy session key or open viewer in a new tab

The navigator supports automatic dark/light mode and a single-column responsive layout for mobile devices.

12.9 Field Navigator

The Field Navigator is an interactive HTML page generated by `visualether analyze`. It displays every field found in a PCAP as a searchable tree with example values and auto-derived regex patterns for

FXT template authoring (Figure 10). The screenshot below uses `moto_edge_30_pro.pcap` from the bundled `5g-nr-radio` sample (see Section 4).

```
visualether analyze moto_edge_30_pro.pcap
```

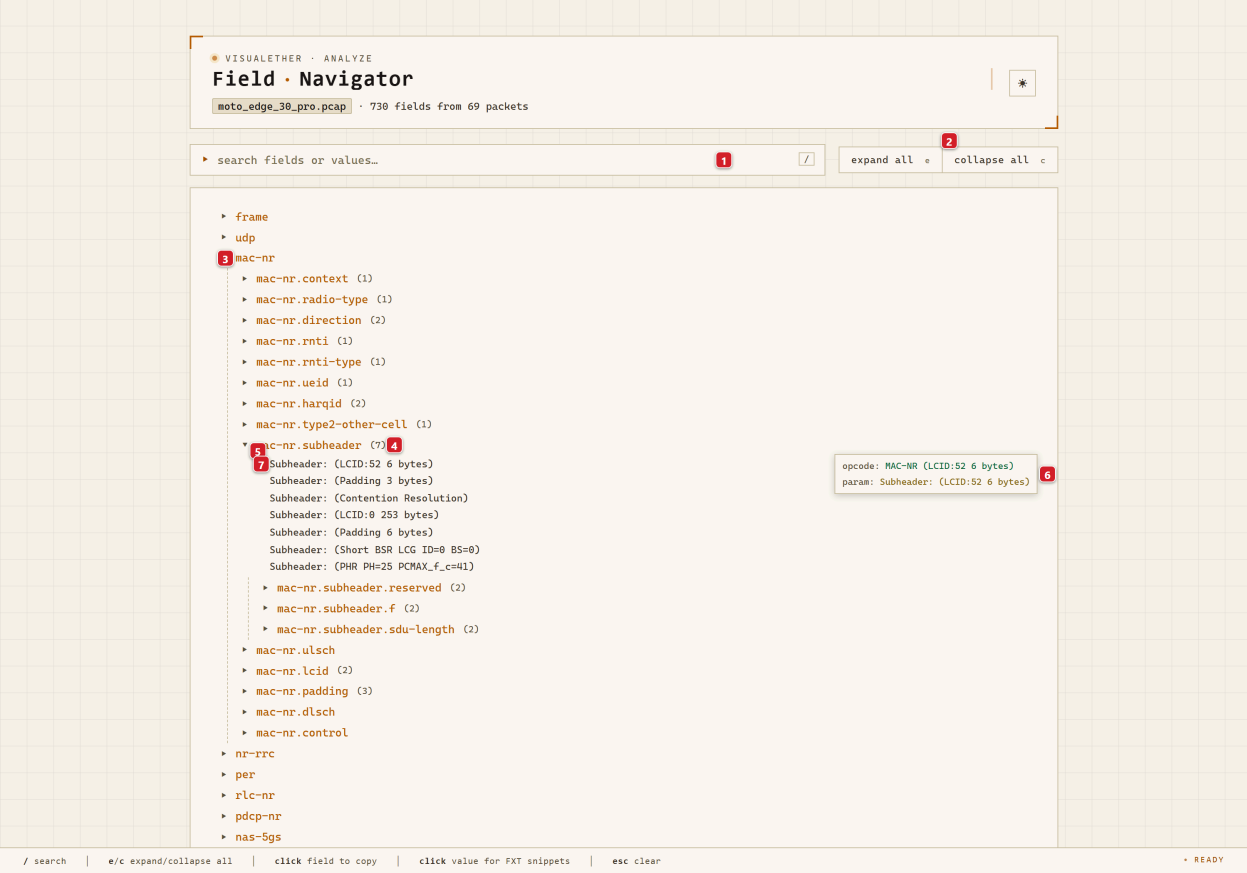

Figure 10: Field Navigator — protocol tree with example values and ready-made FXT snippets.

1. Search input — press `/` to focus; matches field names and example values
2. Expand All / Collapse All (Ctrl+Shift+E / Ctrl+Shift+C)
3. Protocol tree — fields organized by protocol prefix
4. Example count badge — distinct example values discovered
5. Example values — up to 10 per field, near-duplicates collapsed
6. Hover preview — derived opcode and param transformations (green = computed, yellow = passthrough)
7. Click any example value to open the copy menu

When you copy an opcode or param snippet from the Field Navigator, the snippet already includes the `match` and `replace` regex needed to clean up the verbose Wireshark text into a short label suitable for a sequence diagram. Paste the snippet straight into your FXT — no regex authoring required.

> **Tip**

> Use `--protocols bgp,tcp` to limit the Field Navigator to specific protocols when working with multi-protocol captures.

12.10 NDJSON Output

NDJSON is the recurring CI/CD artifact — the setup-versus-runtime split that makes it one (templates and hosts prepared once, NDJSON regenerated per run) is described in Section 2. One JSON object per line, with timestamp, source, destination, opcode, parameters, and session information. NDJSON is also the backing format for lazy mode — `visualether serve` reads it to render PDFs on demand.

12.11 Timestamp Formats

Control how timestamps appear with `--timestamp-format`:

Value	Format
`absolute` (default)	RFC3339, e.g. `2026-01-06T10:30:45.123456Z`. Best for log correlation.
`relative-first`	Elapsed since the first message, e.g. `0.000us`, `1.234s`, `2m 15.500s`. Best for overall timeline.
`relative-previous`	Delta from the previous message, e.g. `+3.932ms`, `+45.000us`. Best for spotting per-step latency.

12.11.1 Wireshark Timestamp Fields for `<remark>`

A `<remark>` template can reference any Wireshark timestamp field:

- `frame.time` — absolute wall-clock time
- `frame.time_relative` — elapsed since the first frame
- `frame.time_delta` — delta from the previous frame
- `frame.time_delta_displayed` — delta respecting display filters
- `frame.time_epoch` — Unix epoch seconds
- `frame.time_utc` — absolute UTC time

Combine with `match` / `replace` to strip verbose prefixes and append explanatory text. See Section 15.11 for a worked example.

12.12 Resetting the Output Directory

Switching formats against an existing output directory leaves the previous run's artifacts in place. The Capture Atlas applies a priority order when surfacing each directory's entry (Session Navigator beats Combined viewer beats HTML Flow Dossier beats PDF), so a stale `_viewer.html` from an earlier combined run can hide a freshly-generated HTML dossier.

Wipe the directory between runs with `visualether clean`:

```
visualether clean              # uses [clean].output from visualether.toml
visualether clean -o output/   # or specify the directory explicitly
visualether clean --dry-run    # preview without deleting
```

The command removes only VisualEther output (`*.pdf` , `*.html` , `*.ndjson` , and the sidecar data files that back the interactive viewers); your input captures, FXT templates, `hosts.txt` , and `README.md` are preserved. See the `clean` section in the Section 18 for the complete file classification rules.

12.13 Next Steps

For protocol-specific FXT recipes that drive each of these output formats, see Section 13. For the AI-assisted workflows that consume NDJSON directly, see Section 14 and Section 15.

13 Analyzing a New Protocol

When you have a PCAP for a protocol you have not analyzed before, this chapter is a workflow guide for finding (or writing) the FXT template that fits. For the FXT format itself, see Section 5 through Section 11.

13.1 Where to Start

The fast path is almost always: copy a working sample, then tweak.

1. **List** the bundled samples to find the closest match for your protocol.

   ```
   visualether list
   ```

2. **Bootstrap** a project from that sample.

   ```
   visualether new <name>
   cd <name>
   ```

   This drops a working `explore.fxt.xml`, `sessions.fxt.xml`, `hosts.txt`, and `visualether.toml` into `./<name>/`, ready to run on the sample's PCAP via `visualether generate --serve`.

3. **Tweak** the FXT to match your own capture: replace the sample PCAP, adjust opcode field names, refine session keys, and re-run `visualether generate`.

13.2 When the Shipped Sample Doesn't Fit

VisualEther ships with 80+ samples, but no project ships everything. Three fall-backs, ordered from "works on any edition" to "Professional / Server only":

- **Discover fields with `visualether analyze`** — generates an interactive Field Navigator (Section 12.9) that lists every Wireshark field your PCAP exposes, along with example values and ready-to-paste FXT snippets. Works on Community Edition; use it when you want to author the FXT yourself.
- **Nick Russo PCAP Diagrams** (Section 17) — hand-tuned `explore.fxt.xml` files covering 41 protocol families. Strong starting point when no shipped sample matches; renders cleanly on Community Edition for most captures.
- **Ask Claude Code** (Professional / Server) — the fastest path for a brand-new protocol. With the MCP server installed, ask Claude Code to read your PCAP and generate `explore.fxt.xml`, `sessions.fxt.xml`, and `hosts.txt` for you. See Section 15 for the workflow and prompt patterns.

13.3 Quick Lookup: Cellular Signaling Correlators

The UE-correlator field names for 4G and 5G control-plane protocols are easy to mis-type. Use these as a glance reference, even when you already have a sample to start from:

13.3.1 5G-NR / 5GC

Protocol	Transport	Port	Session Key
NGAP	SCTP	38412	`ngap.RAN_UE_NGAP_ID` + `ngap.AMF_UE_NGAP_ID`
NAS-5GS	(over NGAP)	–	5g-tmsi or `ngap.AMF_UE_NGAP_ID`
F1AP	SCTP	38472	`f1ap.GNB_CU_UE_F1AP_ID` + `f1ap.GNB_DU_UE_F1AP_ID`
E1AP	SCTP	38462	`e1ap.GNB_CU_CP_UE_E1AP_ID` + `e1ap.GNB_CU_UP_UE_E1AP_ID`
XnAP	SCTP	38422	`xnap.NG_RANnodeUEXnAPID`
PFCP	UDP	8805	`pfcp.seid`

13.3.2 LTE / EPC

Protocol	Transport	Port	Session Key
S1AP	SCTP	36412	`s1ap.MME_UE_S1AP_ID` + `s1ap.ENB_UE_S1AP_ID`
NAS-EPS	(over S1AP)	–	`nas-eps.emm.m_tmsi` or `s1ap.MME_UE_S1AP_ID`
X2AP	SCTP	36422	`x2ap.Old_eNB_UE_X2AP_ID` + `x2ap.New_eNB_UE_X2AP_ID`
GTPv2-C	UDP	2123	`gtpv2.teid`
GTP-U	UDP	2152	`gtp.teid`

13.3.3 Radio-Side Captures

The tables above cover **transport-layer** signaling (SCTP/UDP between core network functions). **Radio-side** captures — the MAC / RLC / PDCP / RRC / NAS stack on the air interface — carry their PDUs in a synthetic UDP envelope that Wireshark's per-layer heuristics recover, reaching you either as a `DLT_USER` linktype or inside an ordinary Ethernet → IP → UDP capture. VisualEther auto-applies the right decode hint for both framings across every subcommand, so a flag-less `visualether analyze` / `generate` just works; Section 16.4 has the full framing, configuration, and decrypt reference, and Section 4 is the worked walkthrough. What matters when authoring the FXT is that the per-UE correlator differs from the transport-layer keys above:

Capture	Heuristic dissectors	Per-UE Session Key

5G NR radio	`mac_nr_udp` / `rlc_nr_udp` / `pdcp_nr_udp`	`mac-nr.rnti` → `rlc-nr.ueid` → `pdcp-nr.ueid`
LTE radio	`mac_lte_udp` / `rlc_lte_udp` / `pdcp_lte_udp`	`mac-lte.rnti` → `rlc-lte.ueid` → `pdcp-lte.ueid`

The session-key column shows a fallback chain because each framing strips a layer: `mac.rnti` exists only when MAC is dissected, `rlc.ueid` only when RLC is, and `pdcp.ueid` only when PDCP is. Use the fallback syntax (`<protocol fallback="true">` , see Section 8.4.1) so a single session-type covers all three framings without forking. The bundled `5g-nr-radio` and `4g-lte` samples' `sessions.fxt.xml` already do this — copy them as the starting point for new radio-layer FXTs. When the DRB user plane is ciphered (opaque `pdcp-{nr,lte}:data`), the decrypt recipe lives in Section 4.7.

13.4 Starter Patterns for Protocols Without a Built-In Sample

For protocols not covered by the shipped samples or the Nick Russo collection, here are minimal templates you can copy as starting points. Each uses the generic `<message>` element with explicit `<source>` and `<destination>` because these protocols don't ride over IP.

13.4.1 IEEE 802.15.4 (Zigbee, Thread, 6LoWPAN)

```
<message style="sea-green">
  <opcode match=".*Association Request.*"
          replace="MAC Association Request">wpan.cmd</opcode>
  <source><address>wpan.src64</address></source>
  <destination><address>wpan.dst64</address></destination>
  <param match=".*= Device Type: (.*)"
         replace="Device: $1">wpan.cinfo.device_type</param>
</message>
```

13.4.2 Bluetooth

For Bluetooth Low Energy over-the-air captures, use Link Layer fields:

```
<message style="royal-blue">
  <opcode>btatt.opcode</opcode>
  <source><address>btle.advertising_address</address></source>
  <destination><address>btle.scanning_address</address></destination>
</message>
```

For HCI traces (controller-to-host event log), `bthci_evt.bd_addr` carries only the remote device's address — there is no separate src/dst at this layer. Pin one side as a literal `Host` axis and let the other side pick up the remote address:

```
<message style="royal-blue">
  <opcode>bthci_evt.code</opcode>
  <source><address match=".*" replace="Host">bthci_evt.code</address></source>
  <destination><address>bthci_evt.bd_addr</address></destination>
</message>
```

13.4.3 CAN Bus

CAN frames carry a message ID, not source / destination addresses. Render senders by `can.id` (the message identifier doubles as a logical sender) and pin the destination axis as `Bus`:

```
<message style="tangerine">
  <opcode>can.id</opcode>
  <source><address>can.id</address></source>
  <destination>
    <address match=".*" replace="Bus">can.id</address>
  </destination>
</message>
```

13.4.4 Layer 2 Protocols (IS-IS, LLDP, STP)

For Layer 2 protocols carried directly over Ethernet, use `<mac-message>` — it automatically resolves axes from `eth.src` / `eth.dst`:

```
<mac-message style="emerald">
  <opcode>isis.type</opcode>
  <param>isis.area</param>
</mac-message>
```

13.5 gRPC / Protobuf

For gRPC captures, key per-RPC sessions on the TCP 4-tuple plus `http2.streamid`, since HTTP/2 multiplexes multiple RPCs over a single TCP connection. When a single TCP segment carries frames for multiple HTTP/2 streams, VisualEther automatically splits multi-value key fields and generates the correct per-stream session keys (see Section 8).

Use `pbf.*` fields for protobuf field extraction (e.g., `pbf.tutorial.Person.name`) instead of generic `protobuf.field.name`.

> **Important**
>
> Wireshark requirement: enable “Dissect Protobuf fields as Wireshark fields” in Edit → Preferences → Protocols → ProtoBuf.

When the `pbf.*` fields are not available, Wireshark exposes protobuf data as generic `protobuf.field.name` and `protobuf.field.value` pairs. Use the name-value param form to combine these into a single parameter row:

```
<param>
  <name>protobuf.field.name</name>
  <value>protobuf.field.value</value>
</param>
```

13.6 Next Steps

For the AI-assisted path that generates FXT templates straight from a PCAP, see Section 15. For idiomatic real-world templates covering 40+ protocol families to study or adapt, see Section 17.

14 AI-Assisted Analysis

VisualEther includes a Model Context Protocol (MCP) server that lets AI assistants analyze packet captures and generate sequence diagrams on your behalf. You describe what you want in natural language, and the AI agent drives the process — automatically selecting tools, generating templates, and producing diagrams.

14.1 Setup

Use the `visualether mcp install` command to register the MCP server with Claude Code. The `--scope` flag controls where the configuration is written:

14.1.1 Project Scope (recommended for team projects)

Registers the server in `.mcp.json` in the current working directory. You can commit this file to version control so the whole team shares the same configuration.

```
visualether mcp install --scope project
```

14.1.2 User Scope

Registers the server in `~/.claude.json`, making it available across all projects without any per-project configuration.

```
visualether mcp install --scope user
```

14.1.3 Upgrading

If the VisualEther executable path changes, uninstall and reinstall the MCP server so the registered path is updated:

```
visualether mcp uninstall --scope user
visualether mcp install --scope user
```

If you upgrade in place (for example, by installing a new MSI to the same directory), no MCP reconfiguration is needed — the existing entry already points to the updated binary.

14.1.4 Removing the Server

To unregister the MCP server:

```
visualether mcp uninstall --scope project   # remove from .mcp.json
visualether mcp uninstall --scope user      # remove from ~/.claude.json
```

> **Tip**
>
> Use project scope when working in a team so the configuration can be shared through version control. Use user scope for personal use across all projects.

14.1.5 Other MCP Clients

The `mcp install` command targets Claude Code, but the MCP server itself speaks standard MCP over stdio, so any MCP-capable client can drive it — Cursor, Windsurf, VS Code, Codex CLI, Gemini CLI, Claude Desktop, and Zed, among others. Run `visualether mcp config` to print the launch command together with the absolute path to your VisualEther executable:

```
visualether mcp config
```

Each client stores its MCP configuration in a different file and format. For copy-paste setup blocks per client, see eventhelix.com/visualether/mcp-clients. Claude Code is the tested-and-supported client; the others follow each tool's own published configuration, so check your client's documentation if a step does not match.

14.1.6 Configuring Fonts for Non-Latin Scripts

If your diagrams include non-Latin text (Devanagari, CJK, Arabic, Hebrew), set the `VISUALETHER_FONTS_DIR` environment variable in your MCP server configuration. The MCP subprocess inherits this variable, so generated PDFs automatically use the specified fonts.

Add the `env` block to the `visualether` server entry — either in `~/.claude.json` (user scope) or `.mcp.json` (project scope):

```
"visualether": {
  "type": "stdio",
  "command": "/path/to/visualether",
  "args": ["mcp", "server"],
  "env": {
    "VISUALETHER_FONTS_DIR": "/path/to/fonts"
  }
}
```

For user scope, this goes under the top-level `"mcpServers"` key in `~/.claude.json`. For project scope, place it in `.mcp.json` in your project directory so team members share the same font configuration through version control.

14.1.7 Environment Variables

The MCP server subprocess inherits environment variables from the parent process. You can set these in the `"env"` block of your MCP server configuration:

Variable	Description
`VISUALETHER_TSHARK_PATH`	Path to `tshark` when it is not on the system `PATH`.
`VISUALETHER_HOSTS_FILE`	Default hosts file for IP-to-hostname substitution.
`VISUALETHER_FONTS_DIR`	Directory of font files for non-Latin scripts (Devanagari, CJK, Arabic, Hebrew) in PDF output.
`VISUALETHER_LICENSE_FILE`	Path to a license file at a non-standard location.

See Section 18.14 for the full list, usage examples, and precedence rules.

14.2 Common Workflows

Most jobs are one prompt. Claude Code drives the rest — protocol detection, FXT selection or generation, rendering, validation, and browser opening. Examples:

Prompt	What Claude Code does
"Generate a sequence diagram from capture.pcap"	Detects protocols, selects or generates an FXT template, renders the diagram, and opens the Capture Atlas in your browser.
"Show me which BGP sessions failed in capture.pcap"	Renders per-session diagrams grouped by outcome, then summarizes the failure patterns and anomalies.
"Generate a hosts file from capture.pcap with descriptive names"	Infers entity roles from the traffic and writes meaningful names. Re-run the diagram to pick the new names up.
"Generate a diagram from all .pcap files in the captures folder"	Accepts glob patterns (`*.pcap`, `capture_*.pcapng`); for split captures, also lets you set a session timeout (default 30 s) for cross-file boundaries.
"Analyze capture.pcap using the TLS keylog file at keys.txt"	Passes the right `tshark` options for TLS / SSL key logs (or a keytab for Kerberos, or a display filter to focus on a subset of traffic).

> **Tip**
>
> Claude Code always extracts all session outcomes for full visibility. Use `<session-filter>` (see Section 9.5) in FXT templates only for CLI / CI/CD filtering, not for MCP workflows.

Template generation works particularly well for well-known protocols (SIP, BGP, HTTP, DNS), telecom signaling (Diameter, S1AP, NGAP), and industrial protocols (Modbus, DNP3, EtherNet/IP) — anywhere Claude Code has domain knowledge. See Section 15 for the FXT-creation walkthrough and in-depth examples (session comparison, state-machine diagnosis, latency analysis, cross-interface correlation).

14.3 Browser Auto-Open via Background `visualether serve`

You don't need to start a web server yourself. When Claude Code produces a diagram, the MCP server starts `visualether serve` quietly in the background on a free local port and opens the Capture Atlas (`http://127.0.0.1:PORT/`) in your default browser — one landing page per project that lists every capture, rather than a separate tab per pcap.

Claude Code targets a shared output directory per project (typically `<pcap-dir>/output`), so the same background serve is reused across follow-up requests — your open browser tabs and any in-progress PDF renders keep working.

What to expect:

- **Capture Atlas landing page.** The page opened at `/` is always the Capture Atlas — a tree view that links to every diagram below it, whether you rendered one capture or a whole folder. Each capture's own viewer is one click into the tree.
- **Live updates.** New per-pcap navigators added by later requests appear on the next refresh; on-demand PDFs render correctly without any restart.
- **Switching captures.** A request pointed at an unrelated directory replaces the auto-spawned serve with a new one; tabs on the old URL go connection-refused — just reload to follow the new URL. If you have started your own `visualether serve` on a directory that already covers the new output, that one is reused instead — it is never replaced or stopped (see Section 18).
- **Stopping it.** The background serve is stopped automatically when the MCP session ends. If the MCP itself is killed forcefully, the next MCP session cleans up the leftover serve and starts fresh. To free the port immediately, use `visualether serve stop` (and `visualether serve list` to see what is running; `visualether serve stop --all` to stop your own serves too — see Section 18).
- **Multiple windows.** Each MCP session uses its own free port, so running Claude Code in two windows side by side does not cause a port collision.

To suppress the browser open and just receive the URL in the result (for scripted workflows), pass `auto_open: false` in the call.

14.4 Next Steps

Section 15 covers the in-depth Claude Code workflows — session comparison, state-machine diagnosis, latency analysis, cross-interface correlation, localized diagrams, and remark annotation — with worked prompt examples for each.

15 Debugging with Claude Code

VisualEther's MCP server turns Claude Code from a text-only assistant into a protocol-aware debugging partner. The CI/CD setup-versus-runtime pattern — `sessions.fxt.xml` and `hosts.txt` prepared once, NDJSON regenerated per run — is described in Section 2.

15.1 Why VisualEther + Claude Code?

The fundamental challenge of using an LLM for packet analysis is scale. A typical PCAP contains thousands of packets with dozens of fields per layer — feeding raw packet data to an LLM exceeds the context window and buries the signal in noise.

VisualEther curates and structures the data before the AI sees it:

1. **FXT templates** extract only the protocol fields you care about — opcodes, session identifiers, status codes, key parameters.
2. **NDJSON output** is one compact JSON object per message, structured for machine consumption.
3. **Session extraction** splits the capture into individual sessions organized by outcome — Claude Code reads only the NDJSON for the sessions it wants to analyze.

The result is kilobytes of curated data instead of megabytes of raw logs.

15.2 Choosing the Right FXT Template

Template	When to Use	Example
`explore.fxt.xml`	Debugging a single session or small interaction. Produces one combined diagram with all messages.	*"Why did this SIP call fail?" — focus on one call's message flow end to end.*
`sessions.fxt.xml`	Analyzing a CI/CD run or production capture with many sessions. Groups messages into individual sessions classified by outcome.	*"How many BGP sessions failed in last night's regression run?" — success/failure/incomplete counts across hundreds of sessions.*

> **Tip**
>
> Ask Claude Code to create both templates at once. Use `explore.fxt.xml` first to understand the protocol flow, then switch to `sessions.fxt.xml` when you need per-session analysis.

15.3 Creating FXT Templates from a PCAP

A common starting point is to ask Claude Code to create FXT templates directly from a PCAP. In CI/CD workflows, this is a setup step rather than a daily one.

```
Please create explore.fxt.xml and sessions.fxt.xml from capture.pcap
```

Behind the scenes, Claude Code:

1. Calls `analyze_capture` to detect protocols.
2. Calls `extract_fields` to inspect field names and example values.
3. Calls `extract_endpoints` to discover IPv4 / IPv6 addresses for the hosts file.
4. Generates both templates using its protocol domain knowledge.
5. Calls `validate_fxt` against the PCAP and iterates if regex patterns or field names need fixing.
6. Calls `materialize_config` to finalize the project — writes `visualether.toml` (with the right `tshark-args` for the capture's linktype, the detected `hosts-file`, and the output directory) and a matching `.gitignore`. After this step `visualether generate` works in the project directory with no command-line flags.

This works particularly well anywhere Claude Code has protocol domain knowledge — see Section 14 for the families that are best supported.

15.3.1 Supplying Decryption Keys to Claude Code

If your capture needs decryption material to be useful — a TLS keylog file for HTTPS / HTTP/2 / QUIC, or a Kerberos keytab for AD-authenticated traffic — mention the file path in the same prompt that asks for the FXTs:

```
Please create explore.fxt.xml and sessions.fxt.xml from @capture.pcap.
The TLS session keys are at @sslkeys.log.
```

```
Please create FXTs from @kerberos-traffic.pcap using the keytab @816.keytab.
```

Claude Code forwards the path to `materialize_config`, which embeds it in `visualether.toml` as the appropriate tshark preference (`-o tls.keylog_file:<path>` or `-o kerberos.decrypt:TRUE -o kerberos.file:<path>`). Subsequent `visualether generate` calls then decrypt the inner protocol traffic automatically — no per-call shell escaping, no environment variables to remember.

15.3.2 What Claude Code Already Knows About VisualEther

The MCP server ships Claude Code a set of VisualEther-specific conventions, so the FXT it generates follows the same patterns the bundled samples use — without you having to spell any of this out:

- **Lightbulb remarks.** `<remark>` annotations use the canonical 💡 `[Frame N] …` shape, keyed by `frame.number`, so they fire on every packet and provide a Wireshark anchor.
- **`multi-match="true"`** on sessions FXTs (see Section 5.2), so TCP SYN/FIN/RST templates fire alongside app-layer templates and properly open/close protocol-specific sessions.

- **Data-plane noise filtered.** High-volume protocols are automatically filtered (`filter="true"` ; see Section 6) so the signaling stays readable. If you want a particular high-volume protocol back in the diagram, ask Claude Code to leave it unfiltered.
- **IPv4 + IPv6 paired.** TCP-based templates get both `tcp-message` and `tcpv6-message` ; UDP-based get both `udp-message` and `udpv6-message` . Session types are shared when the key is protocol-level (e.g., `dns.id` , `diameter.Session-Id`).
- **Hosts file conventions.** Uses `extract_endpoints` for the full IPv4 + IPv6 list; names entities with child.parent dot notation (`AMF.5GC` , `Gi0.R1`); maps every address of one device (IPv4, IPv6, MAC) to the same name so axes merge.
- **Existing files first.** Before generating, looks for `explore.fxt.xml` / `sessions.fxt.xml` in the PCAP directory and reuses them. New files are written next to the PCAP, not in some hidden location.
- **Live-reload aware.** Won't tell you to refresh the browser — the served viewer auto-reloads within a few seconds of regeneration.

If a generated template doesn't follow one of these conventions, ask Claude Code to fix it — the MCP instructions are explicit, and follow-ups are usually one prompt.

15.4 Comparing Successful and Failed Sessions

Once `extract_sessions` has produced per-outcome NDJSON files, Claude Code can read across them and surface what differs:

```
Please extract sessions from sip-registration.pcap and compare
a successful registration with a failed one.
```

Claude Code reads one session from `success/` and one from `failure/` , then explains the divergence in the protocol exchange — typically a missing credential, an unexpected response code, or a dropped retransmission — and suggests where to look next.

15.5 Verifying Feature Behavior

When you want to confirm that a feature works end-to-end, point Claude Code at the capture and the expected behavior:

```
Please verify that BGP peering is established correctly in bgp-test.pcap
Please verify that all HTTP/2 requests in api-test.pcap get proper responses
Please verify that recursive DNS resolution works correctly in dns.pcap
```

Claude Code generates the templates if needed, runs `explore` or `extract_sessions` , reads the NDJSON, and reports either a clean pass (with the observed flow as evidence) or the specific anomaly it found.

15.6 Cross-Interface Comparison

When you have captures from two vantage points (client / server, UAC / proxy, two backends behind a load balancer), Claude Code can correlate messages across them to surface lost packets, latency through middleware, header rewrites, or load asymmetry:

```
I have client.pcap and server.pcap. Please correlate the requests
and tell me whether any client-side requests failed to reach the server.
```

The output identifies messages present on one side but absent (or modified) on the other, and ties the gap to a specific time window or correlator (e.g.
a TCP retransmission burst, a Call-ID, an IP-affinity hash).

15.7 Debugging Protocol State Machines

Claude Code can trace a session's NDJSON against the expected state machine for the protocol and report which transitions failed and why:

```
I have many incomplete Diameter sessions. Please analyze the incomplete
sessions and tell me why they aren't completing.
```

The result is grouped by failure pattern (peer overload, routing change, capture truncation, etc.) so you can address each cluster independently rather than one session at a time.

15.8 Analyzing Timing and Latency

NDJSON includes precise timestamps for every message. Claude Code can identify slow exchanges, correlate them with concurrent activity, and surface candidate root causes:

```
Please analyze the session NDJSON and identify any responses
that took longer than 500ms.
```

> **Tip**
>
> You can also ask Claude Code to add a `max-duration` (see Section 9) to the relevant `<session-type>` declaration. Sessions that respond after the deadline are then classified as `late`; sessions with no response at all are classified as `timeout`. Ask Claude Code to analyze just those buckets to focus on the slow / dropped traffic.

15.9 Batch Analysis Across Multiple Captures

VisualEther accepts glob patterns, so you can analyze a whole test suite in one prompt and get a per-session-type outcome table back:

```
Please extract sessions from all captures in test-captures/ and summarize
the pass/fail distribution.
```

For **ring-buffer / split captures** (`tcpdump -W` , `dumpcap -b` , Wireshark file sets) the files together form one logical capture, so sessions cross the file boundaries. Tell Claude Code to treat them as one:

```
Please extract sessions from capture_*.pcapng (these are dumpcap
ring-buffer splits, treat them as one capture).
```

Claude Code usually recognizes ring-buffer naming patterns and automatically sets `merge_inputs` ; if the pattern is ambiguous, it asks before merging. The CLI equivalent is `--merge-inputs` — see Section 18 for the continuity rules.

For very large session-heavy runs, the default `auto` format already picks `lazy` , so the session index loads instantly (see Section 12). When no human is going to open the viewer, you can also ask Claude Code to skip the HTML detail tree entirely with `--format pdf --format ndjson` .

15.10 Generating Localized and Translated Diagrams

FXT's `replace` attribute embeds arbitrary literal text in message labels — including translations and emoji (see Section 7.9 for the underlying technique). Claude Code picks contextually appropriate emoji and produces natural translations for protocol terminology:

```
Please create explore.fxt.xml and sessions.fxt.xml from profinet.pcap
with Japanese translations and a suitable emoji for each message.
```

The resulting templates carry labels like `📎 Connect / 接続` , `📝 Write / 書き込み` , `⚠ Alarm Error / アラームエラー` . For multi-language teams, ask for parallel files (`explore-ja.fxt.xml` , `explore-ko.fxt.xml` , etc.) — they share the same `match` patterns and only differ in `replace` text.

> **Tip**
>
> Configure `VISUALETHER_FONTS_DIR` for non-Latin scripts. Without the right fonts, CJK / Devanagari / Arabic / Hebrew text won't render in PDF output. See Section 14.1.

15.11 Annotating Diagrams with Remarks

`<remark>` annotations turn a raw packet trace into a self-documenting design diagram — each message carries a one-line explanation of its role in the protocol flow. Ask Claude Code to add them:

```
Please enhance explore.fxt.xml to add remarks that explain the role of
each message in the protocol flow.
```

Claude Code uses the canonical 💡 `[Frame N] …` shape so every annotation includes a Wireshark anchor, and writes the explanation per message type based on its protocol domain knowledge. The result is particularly valuable for protocols you're less familiar with — the diagram doubles as a learning resource.

15.12 Tips for Effective Prompts

15.12.1 Be Specific About What to Analyze

Guide Claude Code toward the information you actually need:

Less Effective	More Effective
"Analyze this PCAP"	*"Extract SIP sessions and summarize why calls are failing"*
"What's in this capture?"	*"Create FXT templates for the Diameter and GTPv2 traffic in this capture"*
"Debug the issue"	*"Compare the successful and failed BGP sessions to find why peering drops"*
"Check the traffic"	*"Verify that all DNS queries get responses within 100ms"*

15.12.2 Iterate with Follow-Up Questions

After an initial pass, drill in:

```
> Please extract sessions from diameter.pcap
> The 5 incomplete sessions concern me. Read those NDJSON files
  and tell me the last message in each.
> Are those all going to the same Diameter peer?
```

Each follow-up reuses the existing NDJSON; Claude Code doesn't have to regenerate the templates or re-run extract.

15.12.3 Use Hosts Files for Clearer Analysis

Friendly entity names make Claude Code's commentary readable. Hosts files also support emoji and bilingual names, so axis headers can carry the same treatment as message labels:

```
Please create a hosts file from capture.pcap with Japanese translations
and emoji prefixes for each device role.
```

When IPv4, IPv6, and MAC addresses for the same device are mapped to the same display name, the diagram axes merge — one column per entity instead of one per address.

15.13 Case Studies

Real Claude Code + VisualEther sessions are published as transcripts at eventhelix.com/visualether/case-studies/. Each case study captures the actual prompts, the tools Claude Code chose, the tables and findings it returned, and the iterative back-and-forth on the way to a diagnosis. They are a useful complement to this chapter — the chapter shows the patterns; the case studies show the patterns running on real captures end-to-end.

15.14 Next Steps

When a Claude Code session surfaces a problem — a template that won't match, an empty diagram, or a radio capture that won't decode — Section 16 collects the common failure modes and their fixes. For idiomatic FXT examples covering 40+ protocols to draw on when authoring templates for the AI to use, see Section 17.

16 Troubleshooting

When a diagram does not look the way you expected, the fastest path is usually to scan the table below first — most surprises in FXT authoring fall into a handful of recurring shapes. The sections after the table cover the deeper categories (multi-input continuity, regex debugging, empty results, undetected radio protocols, performance) with worked examples.

16.1 Common Issues

Problem	Cause	Solution
No messages extracted	Opcode field name wrong	Check Wireshark field names
Duplicate sessions (A->B and B->A as separate threads)	`direction-agnostic="false"` is set when bidirectional grouping is wanted	Remove the attribute — `direction-agnostic` defaults to `true`
Sessions never end	No stop message	Add `session-stop="true"` message
Wrong packets in session	Qualify too broad	Add port/value constraints or `<not-exists>`
Missing fields in output	Field not in capture	Use `<fallback-key>`
Generic template matches first	Template ordering issue	Use `priority` attribute
Fallback matches unwanted packets	Higher-layer protocol present	Add `<not-exists>` condition
False timeouts on multiplexed protocols	Multi-stream frames produce mismatched keys	Handled automatically — see Section 8 for multi-value key field splitting

16.2 Empty Results (No Messages Found)

If `visualether generate` produces no output, investigate the problem systematically:

1. **Verify protocol detection:** Run `visualether analyze` to confirm that protocols are being detected. If none are found, the PCAP may be encrypted, truncated, using non-standard ports, or a bare radio / DLT_USER capture (see Section 16.4).
2. **Template mismatch:** Compare the detected protocols with the FXT template you are using. Try a different suggested template from the `analyze` output.

3. **Port or qualify-filter mismatch:** Run `visualether analyze` to inspect the actual port values in the PCAP. Check whether the protocol uses non-standard ports (for example, SIP on 5080 instead of 5060). Review the `<qualify>` elements in the FXT.
4. **Session-tracking issues:** Ensure that `session-start` messages are present in the capture. Messages without a matching session start are discarded. If the capture does not include the start trigger, use `auto-start` to begin a session on the first qualifying message.
5. **Create a custom FXT:** Use `visualether analyze --protocols` to inspect field values, author an FXT template using domain knowledge, and verify it with `visualether generate`.

16.3 Suggested Template Favors Discovery Chatter

If `visualether analyze` suggests a discovery template (mDNS, LLMNR, SSDP, NetBIOS-NS) when you expected your application protocol, the capture most likely swept in ambient LAN broadcast and multicast traffic. An interface-wide capture records every device's background chatter alongside the conversation you care about; each of those protocols then counts toward detection and can outweigh the handful of frames you actually wanted to analyze.

Scope the capture toward the conversation rather than filtering after the fact:

- **At capture time**, narrow the capture filter to the endpoints or ports of interest — for example `host 203.0.113.10 and port 443`, which keeps the DNS lookup and the unicast session while dropping broadcast and multicast discovery noise.
- **For an existing capture**, open it in Wireshark, apply a display filter, and export the matching packets (*File > Export Specified Packets*) to a smaller PCAP before running VisualEther.

Capture and display filters are part of your capture workflow, so apply them with `dumpcap`, `tshark`, or Wireshark — VisualEther reads whatever packets the resulting PCAP contains.

16.4 No LTE/5G-NR Radio Protocols Detected

srsRAN-style 4G/5G captures carry bare radio and signaling PDUs with no Ethernet/IP/SCTP wrapper, so raw `tshark` decodes nothing above the link layer. VisualEther recognizes the framings these captures use and auto-applies the right decode hint inside every subcommand (`analyze`, `generate`, `fields`, `endpoints`, `protocols`) — so in most cases there is nothing to do: `visualether analyze` reports the radio stack and `visualether generate` renders it with no flags. The PDU is packed one of two ways, each with its own hint:

- **Custom Data Link Type (`DLT_USER`).** The PDU sits directly in a custom linktype (`DLT_USER0`..`DLT_USER15`, linktypes 147..162). A `user_dlts` mapping tells Wireshark which dissector that DLT carries — DLT 147 → `mac-lte`, DLT 152 → `ngap`, and so on.
- **UDP envelope.** The PDU is wrapped in a synthetic UDP packet (srsRAN's `0xbeefdead` source/destination-port convention) and a per-layer heuristic dissector recovers it. This envelope can ride a `DLT_USER` linktype (DLT 149) **or** an ordinary Ethernet → IP → UDP capture; either way the hint enables the heuristics rather than mapping a DLT.

If a capture still shows no protocols, the auto-apply did not fire — usually because the framing is outside the recognised set, or because a `tshark-arg` you supplied took precedence. The rest of this section

explains the auto-apply, how to configure the hint yourself, FXT authoring for these captures, and decrypting the user plane.

16.4.1 Automatic Decode Hints

VisualEther injects the right tshark hint inside every tshark-driven subcommand — `generate`, `analyze`, `protocols`, `fields`, `endpoints` — so a bare `visualether analyze --input capture.pcap` (or `generate`, `fields`, …) decodes the radio stack with no `--tshark-arg` flags, and a one-line notice prints when the auto-apply fires so the behaviour is never silent. Two framings are covered:

- **DLT_USER linktypes (147–152):** the matching `user_dlts` mapping is applied, plus the heuristic-enable flags for DLT 149.
- **UDP envelope over ordinary Ethernet/IP/UDP:** when the link type is Ethernet and the payload is an undissected `udp:data`, the L2-over-UDP heuristics (`mac_{lte,nr}_udp` / `rlc_…` / `pdcp_…`) are enabled so the radio stack surfaces — there is no DLT to map.

Explicit `--tshark-arg` flags and any `tshark-arg` in `visualether.toml` always win over the auto-detect — the auto-apply only fires when the mapping or heuristics aren't already present. The table below covers every cell of the role × stack matrix for srsRAN_4G and srsRAN_Project; match your capture's role and stack to the right row to find the DLT mapping that gets applied.

16.4.2 srsRAN DLT_USER Conventions

Stack	Role	Layer	DLT	Decode-as	Notes
4G	UE / eNB	MAC-LTE	147	`mac-lte`	Direct framing, no UDP wrapper
4G	UE	NAS-EPS	148	`nas-eps`	Bare NAS, no S1AP/SCTP
4G + 5G	UE / eNB / gNB	MAC / RLC / PDCP for LTE & NR	149	`udp` + heuristics	`0xbeefdead` UDP envelope — needs the L2-over-UDP heuristics (see below)
4G	eNB / EPC (MME)	S1AP	150	`s1ap`	Bare S1AP, no SCTP/IP
5G	UE	NAS-5GS	151	`nas-5gs`	Bare NAS, no NGAP/SCTP — 5G analog of DLT 148
5G	gNB / 5GC (AMF)	NGAP	152	`ngap`	Bare NGAP, no SCTP/IP — typical of ZMQ-driven srsRAN ↔ Open5GS rigs

The DLT 149 row covers more than one capture shape: the same `0xbeefdead` envelope carries MAC-side captures (`mac-{lte,nr}-pcap`, needs `--enable-heuristic mac_{lte,nr}_udp`), bare-RLC captures (`rlc-{lte,nr}-pcap`, no MAC layer above, needs `rlc_{lte,nr}_udp`), and bare-PDCP

captures (`pdcp-{lte,nr}-pcap` , no MAC/RLC layer above, needs `pdcp_{lte,nr}_udp`). The six heuristics inspect distinct magic, so enabling all of them is a no-op for non-matching pcaps.

16.4.3 Configuring the DLT Mapping

The `uat:user_dlts:` syntax embeds double quotes that have to survive shell parsing, which trips up bash, zsh, PowerShell, and cmd.exe in different ways. To avoid that pain, configure the mapping once in one of two places, and VisualEther / Wireshark will automatically pick it up: per-project (in `visualether.toml`) or system-wide (in the Wireshark UAT).

16.4.3.1 Recommended: `visualether.toml`

`visualether new 4g-lte` (or `new 5g-nr-radio` , `new 5g-srsran-ngap` , `new 5g-attach` , etc.) writes a project skeleton that already includes the right tshark arguments in `visualether.toml` . After that, plain `visualether analyze` and `visualether generate` calls in the project directory pick the args up automatically — no per-call flags, no shell escaping. `visualether analyze` also recommends bundled samples directly when it detects a known protocol set; the recommendation cites the sample by name (`5g-srsran-ngap`), so the suggested `visualether new <name>` invocation works without translation.

If you are adding the mapping to an existing project, the cleanest form is the `tshark-arg` array — one token per element, passed to tshark literally with no shell parsing, so the only escaping you need is TOML's `\"` for the quotes inside each `uat:user_dlts:` value. Put it in a single `[defaults]` block — every tshark-driven subcommand (`generate` , `fields` , `query-fields` , `protocols` , `analyze` , `endpoints` , `init`) inherits from it. Example for a project mixing all six srsRAN DLT_USER conventions plus the six L2-over-UDP heuristics:

```
[defaults]
tshark-arg = [
    "-o", "uat:user_dlts:\"User 0 (DLT=147)\",\"mac-lte\",\"\",\"\",\"\",
\"\"",
    "-o", "uat:user_dlts:\"User 1 (DLT=148)\",\"nas-eps\",\"\",\"\",\"\",
\"\"",
    "-o", "uat:user_dlts:\"User 2 (DLT=149)\",\"udp\",\"\",\"\",\"\",\"\"",
    "-o", "uat:user_dlts:\"User 3 (DLT=150)\",\"s1ap\",\"\",\"\",\"\",\"\"",
    "-o", "uat:user_dlts:\"User 4 (DLT=151)\",\"nas-5gs\",\"\",\"\",\"\",
\"\"",
    "-o", "uat:user_dlts:\"User 5 (DLT=152)\",\"ngap\",\"\",\"\",\"\",\"\"",
    "--enable-heuristic", "mac_lte_udp",
    "--enable-heuristic", "mac_nr_udp",
    "--enable-heuristic", "rlc_lte_udp",
    "--enable-heuristic", "rlc_nr_udp",
    "--enable-heuristic", "pdcp_lte_udp",
    "--enable-heuristic", "pdcp_nr_udp",
]
```

A shell-parsed `tshark-args = '''…'''` string is also accepted (handy when pasting a command line verbatim), but the array avoids the nested-quote gymnastics. Only add a per-command `[generate]` /

`[protocols]` / etc. `tshark-arg` value when you need that command to deviate from the shared defaults — the per-command value overrides `[defaults]`, and CLI flags override both.

16.4.3.2 Alternative: persistent Wireshark `user_dlts` UAT

For mappings you want available system-wide — to all Wireshark / tshark instances regardless of project — add the rows to the Wireshark UAT file directly (`%APPDATA%\Wireshark\user_dlts` on Windows, `~/.config/wireshark/user_dlts` on Linux/macOS):

```
"User 0 (DLT=147)","mac-lte","0","","0",""
"User 1 (DLT=148)","nas-eps","0","","0",""
"User 2 (DLT=149)","udp","0","","0",""
"User 3 (DLT=150)","s1ap","0","","0",""
"User 4 (DLT=151)","nas-5gs","0","","0",""
"User 5 (DLT=152)","ngap","0","","0",""
```

The MAC / RLC / PDCP heuristics still need to be enabled per-call (`--enable-heuristic mac_{lte,nr}_udp`, `rlc_{lte,nr}_udp`, `pdcp_{lte,nr}_udp` — six flags in total, one per layer × generation) since the UAT only handles the DLT-to-dissector mapping. Enable whichever layer matches your capture's framing; enabling all six is safe because each heuristic inspects distinct magic.

16.4.4 FXT Authoring for DLT_USER Captures

DLT_USER captures lack IP/SCTP/Ethernet entirely, so the standard transport-bound message elements (`<sctp-message>`, `<udp-message>`, `<tcp-message>`) won't match. Use the generic `<message>` element with constant axis labels derived via `replace` on a field that exists in every packet (for example, `ngap.procedureCode`, `mac-nr.direction`, or `s1ap.procedureCode`):

```
<message style="sea-green">
  <opcode match=".*" replace="🤝 NGSetupRequest">
    ngap.NGSetupRequest_element
  </opcode>
  <source>
    <address match=".*" replace="gNB">ngap.procedureCode</address>
  </source>
  <destination>
    <address match=".*" replace="AMF">ngap.procedureCode</address>
  </destination>
</message>
```

Both `samples/4g-lte/explore.fxt.xml` (S1AP / NAS-EPS / MAC-LTE) and `samples/5g-nr-radio/explore.fxt.xml` (MAC-NR / RLC-NR / PDCP-NR / NR-RRC) are reference templates for this pattern. For NGAP captures, `samples/5g-attach/explore.fxt.xml` is the right starting point, even though it ships as `<sctp-message>` — swap the element to `<message>` and add the `<source>` / `<destination>` axis labels above.

16.4.5 Modern srsRAN_Project (DLT 252 / Exported-PDU)

srsRAN_Project (the modern 5G gNB) does **not** assign per-protocol DLT_USER numbers for NGAP / F1AP / E1AP / E2AP / GTP-U. Instead, it uses DLT 252 (Wireshark Exported-PDU) with the dissector name embedded in each frame. Recent Wireshark versions auto-dissect those frames without any `user_dlts` mapping, so they never produce the `user_dlt` signal that VisualEther's auto-apply looks for. If a srsRAN_Project capture still doesn't decode, upgrade Wireshark/tshark — the Exported-PDU dissector has been the default since Wireshark 3.x.

16.4.6 Same Framing over Ethernet/IP/UDP (No DLT_USER)

The MAC/RLC/PDCP-over-fake-UDP envelope does not always ride a `DLT_USER` linktype. It is just as common inside an ordinary **Ethernet → IP → UDP** capture (link type `Ethernet`, not `USER N`) — for example Wireshark work item 19757's `r16_mdt_nr_mac.pcap`. Here there is no DLT to map, so the `user_dlts` table is irrelevant; the payload decodes with the L2-over-UDP heuristics **alone** (`--enable-heuristic mac_{lte,nr}_udp` / `rlc_…` / `pdcp_…`).

This case is handled by the same auto-apply described above: a flag-less `visualether analyze` reports the radio stack and suggests `5g-nr-radio` / `4g-lte`, and `visualether generate` renders it — identical to the DLT 149 case, just without the `uat:user_dlts` line.

16.4.7 Decrypting the User Plane (NR / LTE)

After AS Security Mode Complete the DRB user plane is ciphered, so user-plane PDUs show as opaque `pdcp-{nr,lte}:data` and the SDAP QoS-flow header (NR) never appears. When the UE security keys are available, add the decrypt recipe to `visualether.toml`: the `pdcp_{nr,lte}_ue_keys` UAT row(s) (UEId plus the RRC and user-plane cipher keys), `pdcp-{nr,lte}.decipher_userplane:TRUE`, `mac-{nr,lte}.lcid_to_drb_mapping_source:"From configuration protocol"`, and `rlc-{nr,lte}.call_pdcp_for_…_drb` at the PDCP SN length (default 18-bit). The bundled `5g-nr-radio` sample ships a ready-made `r16-mdt-19757.visualether.toml` carrying exactly this recipe (with the public UE keys from Wireshark work item 19757) — copy it as a template and substitute your own keys.

If the user plane still shows as `pdcp-{nr,lte}:data`, re-check the key values and the SN length (try 12-bit if the bearer uses a 12-bit PDCP SN). Once decrypted, NR reveals SDAP (`sdap.qfi`) and inner IP; LTE reveals inner IP directly (no SDAP layer). See Section 4.7 for a worked NR example.

16.5 TCP-Only Output (No Application-Layer Content)

If the output shows only TCP messages (SYN, ACK, FIN) with no application-layer content, the root cause is usually one of the following:

1. **TCP desegmentation disabled:** Many protocols (SCCP/Skinny, MySQL, MongoDB) span multiple TCP segments and require reassembly before the dissector can decode them. Enable desegmentation:

```
visualether generate --fxt template.fxt.xml --input capture.pcap \
  --output out --tshark-arg "-o" --tshark-arg
"tcp.desegment_tcp_streams:TRUE"
```

2. **Non-standard port:** When a protocol runs on a non-default port, `tshark` will not auto-detect it. Use Decode As to map the port to the correct dissector:

   ```
   visualether generate --fxt template.fxt.xml --input capture.pcap \
     --output out --tshark-arg "-d" --tshark-arg "tcp.port==2000,skinny"
   ```

   Common Decode As rules include `tcp.port==5000,http` (HTTP on port 5000), `tcp.port==20000,bmp` (BGP Monitoring Protocol), `tcp.port==27017,mongo` (MongoDB), `tcp.port==6654,openflow` (OpenFlow), and `tcp.port==49,tacplus` (TACACS+).

3. **Template ordering:** Application-layer templates (for example, `http.request.method`, `mysql.opcode`, `skinny.opcode`) must appear **before** the generic TCP fallback (`tcp.flags`) in the FXT file — VisualEther uses first-match-wins.

16.6 Analysis and Timing Templates Render Nothing

If most of the diagram renders but specific **anomaly** or **timing** templates come out empty — TCP expert events (`tcp.analysis.retransmission`, `tcp.analysis.zero_window`, `tcp.analysis.duplicate_ack`, the handshake RTT `tcp.analysis.initial_rtt`) or request-to-response time fields (`dns.time`, `http.time`, `http2.time`, and the `.time` field for IMAP, RPC / NFS, SMB, LDAP, Kerberos, and DCE/RPC) — the capture is being dissected in a single pass.

These fields only populate when `tshark` runs its **two-pass** analysis (`-2`). The second pass scans the whole capture to correlate retransmissions, duplicate ACKs, and request / response pairs. Without `-2`, the events are still in the packets, but the derived fields are blank, so the templates that match them produce nothing.

VisualEther adds `-2` for you in the common cases: `visualether new` / `visualether init` and the AI-assisted workflow write it into `[defaults].tshark-arg` of the generated `visualether.toml` whenever the template set needs it. You only hit the empty-template symptom after removing `-2` from a project's `visualether.toml`, or when you author a custom FXT that references these fields and run it without `-2`.

In that case `visualether generate` warns before rendering, anchored on the first offending template:

```
explore.fxt.xml:26:3: warning: references two-pass-only field(s)
  (tcp.analysis.zero_window, tcp.analysis.retransmission, +11 more) that
  render empty without tshark two-pass — add `-2` to [defaults].tshark-arg
  in visualether.toml (or pass --tshark-arg=-2)  [FXT-TWO-PASS]
```

Fix it either way:

- **Project (recommended):** add `-2` to `[defaults].tshark-arg` in `visualether.toml`, so every run inherits it:

```
[defaults]
tshark-arg = ["-2"]
```

- **One-off:** pass it on the command line — `visualether generate ... --tshark-arg=-2`.

Two-pass analysis is slightly slower on very large captures, which is why it is opt-in rather than always on.

16.7 FXT Validation Errors

VisualEther validates FXT files before processing.

16.7.1 Diagnostic Format

Every validation finding prints on one line in the clang/gcc/rustc shape that editor terminals (VS Code, JetBrains, and most modern terminals) recognize as a clickable jump-to-source link:

```
myproject/explore.fxt.xml:4:5: error: @replace requires @match [FXT-SCHEMA]
```

Fields, left to right: file path, line number, column number, severity (`error`, `warning`, or `info`), the message, and a stable diagnostic code in brackets. Editors group and filter findings by code, so `[FXT-SCHEMA]` issues can be triaged separately from `[FXT-LIFECYCLE]` advisories.

Code	Family
`FXT-XML-PARSE`	XML well-formedness and serde deserialization
`FXT-SCHEMA`	Structural, attribute, or value violations
`FXT-EXTENDS`	Unresolved template inheritance
`FXT-REGEX`	Regex compile errors and capture-group mismatches
`FXT-FIELD`	Field-usage / double-dip conflicts
`FXT-LIFECYCLE`	Session start / stop pairing problems
`FXT-MULTIMATCH`	Multi-match shadowing or catch-all duplication. The cross-protocol overlap variant is suppressed on `explore.fxt.xml` files (overlap is resolved by priority in compact explore diagrams); a separate variant fires when `multi-match="true"` is set on an `explore.fxt.xml` without a documented justification (a known inner/outer encapsulation pair, or 3+ specific templates on the same opcode field)
`FXT-FILTER`	`filter="true"` combined with lifecycle attributes
`FXT-TRANSPORT`	Transport vs opcode-field-prefix mismatch

`FXT-ENCAP-ORDER`	Inner-protocol template placed after outer carrier
`FXT-TWIN`	IPv4 / IPv6 transport twin coverage gap. Templates marked `single-stack="true"` are exempt. Sibling-protocol pairs (`dhcp` ↔ `dhcpv6` , `rip` ↔ `ripng` , `icmp` ↔ `icmpv6` , `igmp` ↔ `mld`) cover for each other even though their `(field, @match)` tuples differ
`FXT-OPCODE-ANCHOR`	Bare numeric opcode pattern without anchors
`FXT-AUTOV6-REDUNDANT`	Explicit v6 template duplicates an auto-v6 twin
`FXT-AUTOV6-SHADOW`	Catch-all template shadowing a later v6 lifecycle
`FXT-V4-ONLY-KEY`	Session key using IPv4-only fields under auto-v6
`FXT-IO`	I/O error reading the FXT source
`FXT-TWO-PASS`	Template references a field that only populates under `tshark` two-pass (`-2`) — a `*.analysis.*` expert event or a `<proto>.time` response-time field. Advisory: `validate_fxt` always reports it; `generate` warns only when `-2` is not in effect. See *Analysis and Timing Templates Render Nothing* above

16.7.2 Common Errors

Error	Cause	Solution
“duplicate session-type name”	Two `<session-type>` with same name	Use unique names
“session-type not defined”	Message references undefined session-type	Add matching `<session-type>`
“source/destination count mismatch”	Different field counts	Match field counts
“qualify field missing constraint”	`<field>` without value/ regex	Add value, range, or regex
“unexpected <qualify> in protocol message”	`<qualify>` inside message template	Move `<qualify>` to `<session-type>` ; see Section 8 for the qualify rules and how to filter inside a template with `match` / `required` .

16.8 `--merge-inputs` Continuity Errors

`--merge-inputs` assumes that the listed `--input` files form one continuous capture (mergecap-style). VisualEther verifies this before merging sessions across files; if the inputs do not look like a continuous capture, the run fails rather than silently stitching unrelated traffic into the same session. The typical fix is to drop `--merge-inputs` so each file is processed independently, or to reorder the `--input` arguments into chronological order.

Error	Cause	Solution
"❌ … starts before the previous file ended (reverse chronological order)"	Later `--input` argument has an earlier first-packet timestamp than the previous file's last packet	Re-order `--input` arguments in capture order, or drop `--merge-inputs`
"❌ … has a Ns gap from the previous file (threshold 300s)"	Gap between consecutive files exceeds 300 seconds — the files are from different capture sessions	Drop `--merge-inputs` to process independently
"❌ … has no packet timestamps; cannot verify continuity"	A non-first file contains no timestamps — typically an empty or truncated capture	Remove the empty file from `--input`, or drop `--merge-inputs`

16.9 Finding Wireshark Field Names

Run `visualether analyze capture.pcap` to open the Field Navigator (Section 12.9) — a searchable tree of every field in the capture with example values and ready-to-paste FXT snippets. Add `--protocols sip,diameter` to narrow the output. Or click a message in the Combined Viewer (Section 12.2) to inspect a single packet's tree directly.

16.10 Regex Debugging

16.10.1 Step-by-Step Approach

1. Use `display="brief"` on the `<opcode>` to see the raw field value
2. Test your regex pattern in an online regex tester such as regex101.com
3. Check XML escaping: `<` must be `&lt;`, `>` must be `&gt;`, and `&` must be `&amp;`
4. Start with a simple pattern and add complexity incrementally
5. Run `visualether generate` with the FXT and PCAP to verify the matches

16.10.2 Common Regex Mistakes

Mistake	Example	Fix
Unescaped XML characters	`match="<tag>"`	`match="<tag>"`
Too-specific pattern	`match="^Type: OPEN Message$"`	`match="^Type: (OPEN) Message"`
Missing anchors	`match="SYN"`	Matches “SYN”, “SYN, ACK”, etc. — may be intentional
Capture group mismatch	`replace="$2"` with one group	Check group count in `match`

16.11 Protocol-Specific Gotchas

A handful of surprises only show up on specific protocols. They have nothing to do with FXT mechanics — they fall out of how the dissector or the wire format itself works.

16.11.1 WiFi 802.11

Two recurring WiFi-template surprises:

- **Block Ack frame names changed between `tshark` versions** — older releases label them differently from current releases, so a regex on the frame name needs both forms.
- **CTS and ACK frames render as self-messages on the receiver axis** — the 802.11 control frame carries only the receiver address, so VisualEther draws sender = receiver.

Both are documented with worked examples in Section 6.

16.11.2 CAPWAP Wired-Side Captures (IP Axis Split)

When analyzing wired-side CAPWAP captures, `tshark` decodes the inner 802.11 frames within the CAPWAP tunnel. The `<wifi-message>` templates match these frames correctly using MAC addresses. However, if you add a UDP Data `<message>` fallback with `ip.src` / `ip.dst` addressing, it creates IP-addressed axes that split from the MAC-addressed `<wifi-message>` axes.

Solution: Do not include a UDP Data `<message>` fallback in WiFi FXT templates. The inner 802.11 frames are already handled by `<wifi-message>` templates. CAPWAP control messages should use `<udp-message>` (which also uses IP addressing but matches the specific `capwap.control.header.message_type` opcode).

16.12 Performance Tips

For large captures or CI/CD pipelines, the levers are:

- **`<session-filter>`** to restrict output to failure / incomplete sessions (see Section 9).
- **`priority` and template ordering** to keep hot-path templates early (see Section 6).
- **`--protocols`** to narrow `visualether analyze` to the protocols you care about (see Section 18).

- `lazy` **format** so per-session PDFs render only on click (see Section 12).

Those levers cut **rendering** time. The other half of the cost is **dissection** — `tshark` parsing and reassembling packets before VisualEther ever sees them — and it is driven by the protocols in the capture, not by file size. A few stream protocols that reassemble large application messages over TCP are far heavier than their byte count suggests: a VNC / RFB capture of only a few megabytes can take minutes to process, because the dissector reassembles and re-scans each framebuffer-update stream as it grows — so a small VNC capture can take longer than a much larger capture of short, independent flows. The public RFB.pcap sample (10 MB) is a good example.

When dissection dominates, shrink the capture before running VisualEther:

- **Narrow to the conversation of interest** — open the capture in Wireshark, apply a display filter, and export the matching packets (*File > Export Specified Packets*) to a smaller PCAP. This is the most effective lever for a single slow stream, because it cuts the number of packets the dissector has to reassemble.
- **Split a long capture into smaller files** — `editcap -i <seconds>` (by time window) or `editcap -c <packets>` (by packet count) produces independent PCAPs you can process separately. This helps most when the capture spans many distinct conversations rather than one long stream.

16.13 Wrong VisualEther Version Runs (`PATH` Shadowing)

If `visualether --version` reports an older release than the one you just installed or upgraded, you almost certainly have more than one `visualether` binary on your `PATH`. The shell runs the **first** match it finds, so a stale copy in a directory that appears earlier on `PATH` shadows the newer one — the install succeeded, it is simply being eclipsed.

This shows up most on macOS, where one machine can accumulate copies from several install methods:

Install method	Typical location
Homebrew tap	`/opt/homebrew/bin/visualether`
macOS package / disk image	`/usr/local/bin/visualether`
Manual copy (extracted archive, scripts)	`~/.local/bin/visualether`

List every copy on your `PATH` in the order the shell searches them — the first line is the one that runs:

```
which -a visualether
```

On Windows, use:

```
where visualether
```

Run each path with `--version` to see which release it is:

```
/usr/local/bin/visualether --version
~/.local/bin/visualether --version
```

Then remove (or overwrite) the stale copy so the current release wins, and refresh the shell's cached command lookup:

```
rm ~/.local/bin/visualether
hash -r
```

`hash -r` makes the current shell forget its cached command paths; opening a new terminal has the same effect. To keep a copy in a personal `bin` directory instead of removing it, overwrite it from the current install — for example `cp /usr/local/bin/visualether ~/.local/bin/`.

> **Important**
>
> A package-manager upgrade — `brew upgrade`, `apt upgrade`, `dnf upgrade`, or `winget upgrade` — only updates the copy that **that** manager installed. A binary you placed by hand in `~/.local/bin` or `/usr/local/bin` is never touched by the package manager, so it keeps reporting its old version until you remove or replace it yourself.

16.14 Next Steps

For protocol-by-protocol authoring guidance — including starter patterns for protocols without a shipped sample — see Section 13. For idiomatic FXT examples covering 40+ protocols, see Section 17.

17 Learning Resources

This chapter points to a curated collection of pre-rendered sequence diagrams and ready-to-use FXT files that pair well with the rest of this manual. If you learn best from examples rather than reference prose, start here.

17.1 The Nick Russo PCAP Diagrams Collection

The collection is built around more than a thousand packet captures that the late Nick Russo assembled over his career as a CCIE-certified network engineer and educator. Nick described his captures as *"job aids"* — compact references that answer questions like *"What does an OSPF adjacency actually look like on the wire?"* or *"How does an LDP session come up?"* — and that framing carries through directly into the VisualEther renderings.

> **Tip**
>
> **Just want to read the diagrams?** Open diagrams.eventhelix.com/nick-russo/ in a browser. Every capture in the collection is pre-rendered and indexed there as an interactive sequence diagram — no install, no clone, no command line. Bookmark it as a study reference.

17.1.1 Three resources

The collection is published across three places, each aimed at a different audience:

Where	What it gives you
diagrams.eventhelix.com/nick-russo/	Pre-rendered HTML + PDF sequence diagrams for the entire collection. Read them directly in the browser. *Start here if you only want to study the diagrams.*
`eventhelix/nick-russo-fxt`	The `explore.fxt.xml` files and `hosts.txt` topology mappings used to render the diagrams above, organized by protocol family. *Start here if you want to read or contribute to the FXT files.*
`njrusmc.net/jobaid` (Wayback Machine)	Archived snapshot of Nick's original job-aid page, with the PCAPs he published himself. *Start here if you want the source PCAPs to render or experiment with locally.* Nick attached *personal use only* terms to many of these captures — respect that when downloading.

Nick's own GitHub profile (`github.com/nickrusso42518`) hosts a substantial body of networking teaching material — labs, automation scripts, CCIE study aids — worth browsing in its own right.

17.1.2 About Nick Russo

Nick Russo (1989–2024) was one of the most prolific networking educators of his generation. He held multiple CCIE certifications, worked across service-provider, security, and data-center disciplines, and authored an extensive catalog of Pluralsight video courses spanning BGP, OSPF, MPLS, IPsec, AWS networking, automation, and the CCIE/CCNP curricula. He also wrote and self-published books on networking topics, contributed open-source automation tooling, and ran njrusmc.net as a free public reference for engineers studying or troubleshooting in the field.

What set Nick's work apart was the combination of breadth and an instinct for what beginners actually struggle with. He explained protocols not as abstract specifications but as observable behavior — *here is the packet, here is what it means, here is why it matters*. The capture library you see referenced throughout this chapter was an extension of that teaching style: every capture exists to answer a specific question that someone studying or operating a network would actually ask.

"Packet captures help reveal the ground truth. Use these PCAPs as references when troubleshooting complex issues."

He passed away in 2024. The collection here, the rendered diagrams at diagrams.eventhelix.com/nick-russo/, and the surviving archives of his work continue his teaching — the same captures, now also available as searchable visual flows that make protocol behavior obvious at a glance. Pull requests against the FXT files that sharpen labels, expand coverage, or fix misleading diagrams are one practical way to keep his contribution growing.

17.1.3 What the collection covers

The FXT repository is organized by protocol family — one directory per protocol, each with a bespoke `explore.fxt.xml` tuned to the matching capture set on Nick's job-aid page. The PCAPs themselves aren't redistributed in the repo (Nick's *personal use only* terms); pull them from the Wayback Machine archive linked above when you want to render locally. At a glance:

Category	Directories	Representative topics
Routing	7	BGP, OSPF, IS-IS, EIGRP, Babel, RIP, static / dynamic interaction
Switching / L2	4	STP / RSTP, VLAN / DTP / LACP, CDP / LLDP
IP Services	6	DNS, DHCP, NAT44 / 64 / 66, ICMP, NTP, IRDP
MPLS / SR	2	LDP, RSVP-TE, Segment Routing + PCEP + BGP-LS + SRv6
Tunneling / VPN	4	IPsec (IKE / ESP / AH), GRE, L2TPv3, VXLAN / Geneve
Multicast	2	PIM (SM / SSM / Bidir), IGMP, MSDP, Auto-RP
Network Management	5	SNMP, Syslog, NetFlow / IPFIX, AAA (RADIUS / TACACS+), gRPC / gNMI telemetry

Applications	6	HTTP / 1-3, TLS, FTP, SMTP / POP3 / IMAP, databases, storage (SMB / NFS / iSCSI)
Collaboration / Media	1	Skinny (SCCP), SIP, H.323, RTP / RTCP
IoT / Wireless	3	CoAP, MQTT, 802.11 management, EAPoL, Meraki cloud-managed APs
SDN	1	OpenFlow controllers

Every directory carries a single `explore.fxt.xml` that is shared across all PCAPs in that folder. That single-template-per-protocol pattern is a deliberate teaching device: the template is small enough to read in one sitting, and the output it produces stays consistent across every capture that exercises the protocol.

17.1.4 Why the templates are worth reading

Even if you never render any of the diagrams yourself, the FXT files in the FXT repository are some of the clearest examples of idiomatic FXT style available. A few patterns are especially worth noticing:

- **Emoji-prefixed opcodes and params** encode state-machine transitions at a glance. Babel's `babel_pcap/explore.fxt.xml` uses 👋 for Hello, 👂 for I-Heard-You, 📣 for Update, 🌐 for prefix, 🔢 for sequence number, 📏 for metric, and 🆔 for router-id. The sequence diagram reads from top to bottom, in the same order you would narrate the protocol.
- **Lightbulb remarks** (💡) embed RFC references and protocol theory directly into the diagram — see the OSPF, BGP, PIM, and LDP templates. The remark appears next to the message in both the HTML viewer and the Markdown export, turning the diagram into self-contained study material.
- **Self-documenting headers**: most templates open with a 30-60 line comment summarizing the state machine, multicast addresses, key field names, and representative captures. The OSPF template is the canonical example; open it alongside Section 6 to see how the attributes described in this manual appear in practice.
- **IPv4 / IPv6 parity**: where a protocol runs over both IP versions, the explore templates rely on the root `auto-v6="true"` attribute (see Section 6.1.1) so a single `<udp-message>` or `<tcp-message>` declaration covers both address families. Session-keyed samples additionally use the synthetic `addr.src` / `addr.dst` fields so the 4-tuple key shape forms identically on v4 and v6 (see Section 8.3.4).

None of the templates relies on session tracking, so they are an ideal first-pass example of what a well-groomed `explore.fxt.xml` looks like.

17.1.5 Using the collection as a learning path

A suggested progression for someone who is new to VisualEther:

1. **Browse the rendered diagrams first.** Open diagrams.eventhelix.com/nick-russo/ and click through a protocol you already understand — OSPF, BGP, IS-IS, anything. Confirm that the visual flow matches your mental model. This is also the fastest sanity check on whether sequence diagrams are useful for the kind of question you are trying to answer.

2. **Read the matching FXT.** Clone the FXT repository (`git clone https://github.com/eventhelix/nick-russo-fxt`) and open the `explore.fxt.xml` for the same protocol you just browsed. Cross-reference what you saw on screen with the FXT attributes (`match`, `replace`, `style`, `bookmark`) that produced it — most map directly to sections of this manual.
3. **Render one protocol locally.** Download the matching capture set from the archived job-aid page (`njrusmc.net/jobaid` on the Wayback Machine), drop the `.pcap` / `.pcapng` files into the matching directory of the FXT clone, and run:

   ```
   visualether generate \
       --fxt ospf_pcap/explore.fxt.xml \
       --input "ospf_pcap/*.pcapng" \
       --hosts-file ospf_pcap/hosts.txt \
       --output output/ospf_pcap \
       --format combined
   ```

   Open the resulting `output/ospf_pcap/<stem>/<stem>.html` to confirm that your local toolchain reproduces what is hosted online.
4. **Modify a template to answer a question.** Pick a capture that looks a little sparse, open that directory's `explore.fxt.xml`, and add a `<param>` for a field you would like to see. Re-render and diff the output. This is one of the fastest ways to internalize the field-extraction syntax covered in Section 7.
5. **Contribute back.** Pull requests against the FXT repository that improve templates, sharpen labels, or fix misleading diagrams are welcome. Include the affected capture name(s) in the PR description so reviewers can spot-check against the matching diagrams on `diagrams.eventhelix.com/nick-russo/`.

> **Tip**
>
> Most captures render cleanly on the free **Community Edition** of VisualEther — no license key required. The CE limits still apply (explore FXT only, up to 10 diagram axes per PDF, PDF output only), but the captures are intentionally small teaching aids, so the overwhelming majority stay well within those limits. Use the Community Edition if you simply want to follow along; use the Professional Edition when you need the interactive combined viewer, sessions mode, or captures with diagrams that require more than 10 axes.

17.1.6 Contributing FXT improvements

The FXT repository accepts pull requests against `explore.fxt.xml` and `hosts.txt`. A few things that make a PR easy to merge:

- **Reference the affected captures.** List which `<protocol>_pcap/<capture>.pcapng` you tested against (download from the archived job-aid page at `njrusmc.net/jobaid` on the Wayback Machine) and what changed in the rendered diagram.
- **Use topology-role host labels.** `R1 (PE)`, `R2 (RR)`, `SW1 (DR)`, `WLC`, `AP` — prefer roles over raw IP addresses in `hosts.txt`.

- **Keep the FXT readable.** The single-template-per-protocol pattern is a deliberate teaching device; resist the urge to factor out shared snippets unless they meaningfully reduce noise.
- **Confirm Community Edition still renders.** Diagrams that would exceed 10 axes are refused under Community Edition (see Section 12); that's fine for most captures, but worth knowing.

The packet captures themselves are not redistributed in the FXT repository — they are governed by Nick's original *personal use only* terms. PR contributors who need a capture for testing should pull it from the archived job-aid page on the Wayback Machine (linked above).

17.2 Other pointers

- **Built-in sample catalog**: VisualEther ships with 80+ sample protocols that `visualether new <protocol>` can pull into a new project for you. Those samples are the conservative, well-tested baseline; the Nick Russo collection complements them with an order of magnitude more captures per protocol. Run `visualether list` to see what is available locally.
- **Analyzing a new protocol**: Section 13 walks through finding a starter template (shipped sample, Nick Russo collection, Claude Code, or `visualether analyze`) and includes starter patterns for non-IP protocols. Keep it side-by-side with the Nick Russo `explore.fxt.xml` files when designing a new template.

18 Appendix: CLI Reference

> **Tip**
>
> This appendix is reference material. Skip it on a first read — the narrative chapters introduce each flag in context, and you can come back here when you need to look one up.

18.1 Commands Overview

Command	Description
`generate`	Generate sequence diagrams from Wireshark PCAP files
`analyze`	Generate an interactive HTML field navigator from a PCAP file
`mcp`	MCP server management (install, config, uninstall, start)
`list`	List built-in protocol samples
`new`	Create a new project directory from a built-in sample
`init`	Initialize the current directory from a built-in sample
`serve`	Serve generated output via HTTP with plain and gzip support
`clean`	Remove VisualEther artifacts from an output directory
`license`	Manage VisualEther license (show, activate, deactivate, verify)
`eula`	Display the End User License Agreement for the active edition
`manual`	Open the VisualEther User Manual
`fields`	Extract a tree of fields from a PCAP file (used by MCP server)
`query-fields`	Dump a frame-indexed `tshark -T fields` matrix for chosen fields (used by MCP server)
`endpoints`	Extract unique endpoints (IP addresses or IP:port pairs) from a PCAP file (used by MCP server)
`protocols`	Extract a list of protocols from a PCAP file (used by MCP server)

18.2 The `generate` Command

Generate sequence diagrams from packet captures.

```
visualether generate [OPTIONS]
```

18.2.1 Options

Option	Description
`-i, --input <FILE>...`	Input files or glob patterns. Supports `*` (any filename), `?` (single character), `[abc]` (character class), and `**` (recursive directory traversal). Examples include `*.pcap`, `capture_*.pcapng`, `**/*.pcapng` (all pcapng files in subdirectories), and `traces/**/test_?.pcap`. If omitted and `visualether.toml` has no `input` either, VisualEther auto-discovers `*.pcap`, `*.pcapng`, `*.pcap.gz`, and `*.pcapng.gz` in the current directory. This zero-argument auto-discovery is top-level only — it does **not** recurse into subdirectories. To include captures in nested folders, pass an explicit recursive glob such as `**/*.pcapng`.
`-f, --fxt <FILE>`	FXT template file (explore or sessions). If omitted and `visualether.toml` has no `fxt` either, `generate` auto-iterates over `explore.fxt.xml` and `sessions.fxt.xml` in the current directory (whichever exist) and renders both. Community Edition skips `sessions.fxt.xml` (sessions are Professional only).
`-o, --output <DIR>`	Output directory for generated files.

18.2.2 Diagram Options

Option	Description
`--axis <LEVEL>`	Axis grouping: `ip` (default) or `port`. Use `port` for loopback / same-host captures (identical source and destination IP, e.g. `127.0.0.1`) — otherwise every endpoint collapses onto a single axis. The MCP tools and `materialize_config` detect this and apply / persist `axis = "port"` automatically; for CLI runs, set it here or via `[generate].axis` in `visualether.toml`. For Layer-2 captures built from `<mac-message>` / `<wifi-message>` templates the diagram is keyed by MAC address and this option has no effect (both values render the same lifelines) — leave it at the default. There is no `mac` level here; the link-layer element type already

	drives the axis. Any value other than `ip` or `port` is rejected with an error listing the accepted values.
`--timestamp-format <FMT>`	Timestamp display format (see below): `absolute` (default), `relative-first`, or `relative-previous`.
`--max-params <N>`	Maximum parameters to display per message. Parameters beyond this limit are summarized with `...` (default: 15, 0 = disable).
`--hosts-file <FILE>`	Hosts file for IP/port to hostname substitution. Overrides `VISUALETHER_HOSTS_FILE` environment variable.

Timestamp Formats:

- `absolute` — Shows the wall-clock time of each packet in RFC 3339 format (for example, `2026-03-10T14:30:05.123456Z`). Best for correlating with logs or other time-referenced data.
- `relative-first` — Shows elapsed time since the first packet in the trace or session (for example, `+0.000000`, `+0.003412`, `+1.250000`). Useful for understanding overall timing from the start of a flow.
- `relative-previous` — Shows the time delta since the previous message (for example, `+0.000000`, `+0.003412`, `+0.001200`). Useful for spotting delays between consecutive messages.

18.2.3 PDF Options

Option	Description
`--fonts-dir <DIR>`	Directory containing additional font files (NotoSans, CJK, RTL). All font files must be placed directly in this directory; subdirectories are not scanned. Overrides the `VISUALETHER_FONTS_DIR` environment variable.
`--multilingual-line-height`	Use taller line height for CJK/RTL scripts.
`--monochrome`	Black & white output for printing.
`--paper-size <SIZE>`	Paper size: `a0` – `a4`, `letter`, `legal`, `tabloid`, `ledger`, or a custom size (for example, `8.5x11in` or `210x297mm`).
`--footer <TEXT>`	One-line page footer for PDF output (Professional / Server). Supports markdown — e.g. `**bold**`, `*italic*`, `[link](https://example.com)`. The placeholder `{page}` is replaced with the page number. Ignored in Community/Trial, which stamp their own fixed edition watermark.

18.2.4 Output Format

Option	Description
`--format <FMT>...`	Output format(s): `auto` (default), `combined`, `pdf`, `html`, `lazy`, `ndjson`, `markdown` (or `md`). `auto` resolves by mode and size: `combined` for ≤15 MB; for >15 MB, sessions mode picks `lazy` (defer per-session PDFs to on-demand) and explore mode picks `pdf` (single diagram, nothing to defer). When several `--input` files are rendered independently (without `--merge-inputs`), `auto` resolves per file from that file's own size, so a wildcard mixing small and large captures renders each at the right fidelity instead of inheriting one batch-wide decision. `lazy` generates NDJSON with a session navigator; PDFs render on demand via `serve`.
`--output-mode <MODE>`	Output mode for programmatic use: `text` (default) or `json`.
`--gzip`	Gzip-compress HTML and external JSON files (Professional / Server). Served outputs are decompressed transparently in the browser by `visualether serve`.
`--serve`	Spawn `visualether serve` for the output directory and open the Capture Atlas (`/`) in your default browser (Professional / Server). When a serve is already bound to that root, the call is a no-op — no duplicate tab is opened. Default: produce the output only; start the server later with `visualether serve <output>`.

> **Tip**
>
> **`generate` is passive by default; `init` and `new` auto-serve.** `init` and `new` spawn `visualether serve` and open the workspace Capture Atlas (the per-session split-pane viewer iframes a sibling PDF, which Firefox/Chrome/Safari block under `file://`; HTTP serving keeps everything same-origin). Plain `generate` only writes files — add `--serve` if you want the same auto-serve + atlas-open after a `generate` run. Community Edition outputs are self-contained PDFs that work directly via `file://`, so no auto-serve runs.

18.2.5 Session Options

Option	Description
`--session-filter <FILTER>`	Override the FXT `<session-filter>` at runtime. This controls which sessions appear in the output based on their

	outcome. Specify a space-delimited list of outcome categories to include: `success` (normally completed), `failure` (error or abnormal termination), `late` (exceeded the `max-duration` threshold but a stop arrived after), `timeout` (a `max-duration` or `idle-timeout` cap was hit with no closing message), and `incomplete` (session never properly closed). Any other value is rejected. For example, `--session-filter "failure incomplete"` shows only problematic sessions, which is useful for CI/CD pipelines.

18.2.6 Batch Processing Options

When you pass multiple `--input` paths to `generate`, VisualEther supports two workflows:

- **Independent mode** (default) — each input is rendered as its own capture in a separate subdirectory under the output directory.
- **Merge mode** (`--merge-inputs`) — all inputs are treated as one logical capture, so sessions can span across files.

With a single input file, the output still goes into a subdirectory named after the capture stem, so the layout stays consistent in all cases.

Option	Description
`--merge-inputs`	Merge all `--input` files into one logical capture. Use this for ring-buffer or split captures that belong to the same traffic stream. VisualEther fronts dissection with `mergecap -w - | tshark -r -`, so stateful dissectors (QUIC/TLS handshake, HTTP/2 stream tables, TCP reassembly) span file boundaries and Wireshark stream indices stay globally unique inside the merged stream. **Important:** files must be listed in chronological order and must form one continuous capture. VisualEther runs a `capinfos` pre-pass and rejects the run if a later file starts before the previous one ends, if the gap between files exceeds 300 seconds, or if a non-first file has no packet timestamps. If that happens, either reorder the inputs or remove `--merge-inputs` and process the files independently.

18.2.7 Advanced Options

Option	Description

`--tshark-args <ARGS>`	Additional arguments passed to `tshark` as a shell-parsed string (for example, `"-o kerberos.decrypt:TRUE"`).
`--tshark-arg <VALUE>`	Additional `tshark` argument passed literally (repeatable). Use this for arguments that contain special characters, such as double quotes. Example: `--tshark-arg -o --tshark-arg "uat:rsa_keys:\"cert.key\",\"\""`

18.2.8 Examples

```
# Project-dir workflow (recommended): generate auto-iterates over
# explore.fxt.xml + sessions.fxt.xml, auto-discovers PCAPs and hosts.txt.
cd my-project
visualether generate --serve

# Basic explicit form: single capture into out/<pcap-stem>/...
visualether generate --fxt template.fxt.xml --input capture.pcap --output out

# Multiple input files (independent mode — one subdir per input)
visualether generate --fxt template.fxt.xml \
  --input part1.pcap --input part2.pcap --output out

# Glob pattern (recursive `**` also supported, e.g. "captures/**/*.pcapng")
visualether generate --fxt template.fxt.xml \
  --input "test_*.pcap" --output out

# Paper size + timestamp format + hosts file (monochrome for printing)
visualether generate --fxt template.fxt.xml --input capture.pcap \
  --output out --paper-size legal --timestamp-format relative-previous \
  --monochrome --hosts-file hosts.txt

# Spawn `visualether serve` and open the Capture Atlas (Professional / Server)
visualether generate --fxt template.fxt.xml --input capture.pcap \
  --output out --serve

# Custom one-line page footer (Professional / Server; markdown + {page}
placeholder)
visualether generate --fxt template.fxt.xml --input capture.pcap \
  --output out --footer "Page {page} | [Acme Corp](https://acme.example)"

# Gzip-compress the on-disk HTML / JSON sidecars (Professional / Server)
visualether generate --fxt template.fxt.xml --input capture.pcap \
  --output out --gzip
```

```
# Lazy mode for large captures (PDFs render on demand via serve)
visualether generate --fxt sessions.fxt.xml --input large.pcap \
  --output out --format lazy

# Override session filter at runtime (e.g. CI/CD failure-only runs)
visualether generate --fxt sessions.fxt.xml --input capture.pcap \
  --output out --session-filter "failure late timeout"

# Pass extra args through to tshark (Decode-As shown; same shape works
# for `-o tcp.desegment_tcp_streams:TRUE` and other -o knobs)
visualether generate --fxt template.fxt.xml --input capture.pcap \
  --output out --tshark-arg "-d" --tshark-arg "tcp.port==2000,skinny"

# Merge mode: ring-buffer pcaps dissected as one logical capture
# via mergecap, so sessions span files. Output goes to
# out/<common-prefix>/... (e.g. ring_*.pcap → out/ring/)
visualether generate --fxt template.fxt.xml --merge-inputs \
  --input "ring_*.pcap" --output out
```

18.3 The `list` Command

List built-in protocol samples:

```
visualether list [OPTIONS]
```

Option	Description
`-o, --output <FILE>`	Output file for Markdown export (displays table if omitted).

18.4 The `new` and `init` Commands

VisualEther provides two cargo-style scaffolding commands for setting up a project from a built-in sample.

18.4.1 `new` — Create a new project directory

```
visualether new <SAMPLE> [OPTIONS]
```

Creates a new directory named `<SAMPLE>` in the current directory and populates it with the chosen sample. The command fails if a directory with that name already exists.

This is the recommended way to start a project — it keeps the sample files contained in their own subdirectory.

After copying the sample, `new` (and `init`) regenerates the explore and sessions diagrams for **every** capture file the sample shipped — not just the first one. Each capture lands in its own per-stem

subdirectory under `output/`, and the browser opens at the served root (`http://127.0.0.1:<port>/`). The Capture Atlas there scans 8 levels deep, so every per-pcap viewer the sample produced is one click in. See the `serve` command below for the Atlas behavior.

`new` anchors the auto-spawned `visualether serve` at the **current working directory**, not at the project's own `output/`. This means two consecutive `visualether new` commands from the same workspace directory share one serve: the second project's per-pcap subdirs appear in the same workspace-wide Capture Atlas on its next refresh, and any browser tabs you still had open on the first project keep working. `init` keeps its per-project anchor since the user has deliberately `cd`'d into the project directory.

18.4.2 `init` — Initialize the current directory

```
visualether init <SAMPLE> [OPTIONS]
```

Initializes the **current directory** from the chosen sample. Useful when you have already created (and `cd`'d into) the directory you want to use. Be aware that this writes sample files directly into the current directory — prefer `new` when you do not want to mix sample content with existing files.

18.4.3 Options (both commands)

Option	Description
`<SAMPLE>`	Built-in sample to use as starting point (positional, required).
`--tshark-args <ARGS>`	Additional arguments passed to `tshark` as a shell-parsed string.
`--tshark-arg <VALUE>`	Additional `tshark` argument passed literally (repeatable).
`--no-serve`	Skip the auto-spawned `visualether serve` and the Capture Atlas browser open. By default, `new` and `init` spawn a serve for the workspace and open `/`; if a serve is already bound to that root, no new tab is opened.
`--footer <TEXT>`	One-line page footer stamped on every PDF that `new` / `init` renders from the scaffolded `<SAMPLE>`'s captures (Professional / Server). Same markdown / `{page}` rules as `generate --footer`. Ignored in Community/Trial.

18.4.4 TOML-to-CLI Mapping

The `[init]` section uses the same option names as the `new` and `init` commands, with kebab-case keys where needed. For example:

TOML Key	CLI Flag

`sample = "bgp"`	Positional `bgp` (ignored for `new`, which requires `<SAMPLE>` on the command line)
`tshark-args`	`--tshark-args`
`tshark-arg = ["-d", "tcp.port==2000,skinny"]`	`--tshark-arg -d --tshark-arg tcp.port==2000,skinny`
`no-serve = true`	`--no-serve`
`footer = "EventHelix VisualEther — page {page}"`	`--footer "..."` (Professional / Server only)

18.4.5 Example

```
visualether new bgp
cd bgp
```

18.5 The `analyze` Command

Generate an interactive HTML field navigator from a PCAP file:

```
visualether analyze <FILE> [OPTIONS]
```

Option	Description
`<FILE>`	Input PCAP/PCAPNG file (required positional).
`-o, --output <FILE>`	Output HTML file path (default: `<stem>_fields.html` next to input).
`--protocols <PROTOCOLS>`	Comma-separated protocol filter (e.g., `bgp,tcp`).
`--full-scan`	Scan the entire capture file instead of stopping early. By default, VisualEther stops scanning after 2000 consecutive packets with no new discoveries, which is fast and sufficient for most captures. Use `--full-scan` when working with diverse captures that contain many different protocols spread throughout the file.
`--tshark-args <ARGS>`	Additional arguments passed to tshark (shell-parsed string).
`--tshark-arg <VALUE>`	Additional tshark argument passed literally (repeatable).
`--no-open`	Do not auto-open the HTML in a browser.

18.6 The `serve` Command

Serve generated output via HTTP. Professional / Server `new` and `init` auto-spawn this server and open a browser tab on the workspace Capture Atlas; `generate` only does so when given `--serve`. You can also run `visualether serve` manually — e.g. after a plain `generate` run — or on any pre-existing output directory.

The command accepts every output shape `generate` (and the MCP `extract_sessions` / `explore` tools) can produce:

- A Professional / Server `generate` directory — HTML navigator + sidecar data files, serves directly.
- A `--format lazy` directory — NDJSON + navigator, with per-session PDFs rendered on demand from the corresponding `.ndjson` source when the user opens a session.
- A Community `generate` directory — single self-contained PDF; serve simply hands it back as a static file.
- A single `*_viewer.html` or `*.pdf` at the root — redirected to from `/`.
- A **parent directory** containing several per-pcap subdirs (each with its own navigator). Serve synthesizes a **Capture Atlas** tree view at `/` — the atlas scans up to 8 directory levels deep — and lazy children within that depth automatically get on-demand PDF rendering. Per-pcap navigators created **after** `serve` started are picked up on the next request — no restart needed.

Gzip-compressed HTML/JSON files (from `--gzip`) are served transparently with the appropriate `Content-Encoding` header.

```
visualether serve <ROOT> [OPTIONS]
```

Option	Description
`<ROOT>`	Directory to serve. Any output of `generate` (web, lazy, or inline) works; a parent collecting multiple per-pcap output directories renders the Capture Atlas. Required unless a management subcommand (`stop` or `list`) is given.
`--port <PORT>`	Port to listen on (default: 3000).
`--bind <ADDR>`	Address to bind to (default: `127.0.0.1`).
`--no-open`	Do not auto-open the browser.
`--idle-timeout <SECS>`	Auto-exit when no HTTP request has been received for this many seconds. Live-reload polls from open browser tabs count as activity, so the server stays alive as long as any tab is open — the timeout only fires once every tab has been closed (or was never opened). Disabled by default, so a serve you start yourself runs until you stop it. The MCP server passes `--idle-timeout 600` to its auto-spawned serve so a forgotten session reaps itself ~10 minutes after the last tab closes; the next MCP tool call transparently respawns the serve if it has exited.

18.6.1 Managing Running Serves

Beyond starting a server, `serve` also manages the serves already running on your machine:

Command	Description
`visualether serve list`	List every VisualEther serve currently running — the background one the MCP server / `generate --serve` auto-spawns, plus any you started with `visualether serve` — each with its port and served directory.
`visualether serve stop`	Stop the auto-spawned background serve and free its port. Serves you started yourself are left running. Safe to run when nothing is there — it does nothing in that case.
`visualether serve stop --all`	Stop **every** VisualEther serve — the auto-spawned one and all the serves you started yourself — to free every port at once.

Before ending a process, each stop confirms that the program on the recorded port really is a VisualEther serve, so an unrelated program that happens to reuse the same port is never touched.

A manual serve as an anchor. Start `visualether serve` on a directory that holds (or will hold) several captures and it becomes the landing page for that whole tree. When you afterwards render a capture **beneath** it — via `generate --serve` or a Claude Code analysis — VisualEther sees that the anchor already covers the new output, refreshes that tab in place, and does not open a second tab or start a second serve. Your serves are tracked separately from the auto-spawned one, so a later render on an unrelated directory never disturbs a serve you are monitoring.

If the MCP server crashes or is force-killed while its background serve is running, you have two ways to recover:

- **Do nothing.** The next time the MCP server starts, it cleans up the leftover serve and begins a fresh session. No manual steps required.
- **Free the port immediately.** Run `visualether serve stop` to shut down the leftover serve without restarting the MCP session or opening Task Manager.

18.6.2 Examples

```
# Plain `generate` doesn't auto-serve — bring up the server manually
# (or rerun `generate --serve` and let it spawn the server for you).
visualether serve out --port 8080
```

```
# Generate lazy output for a large session-based capture
visualether generate --fxt sessions.fxt.xml --input large.pcap \
  --output out --format lazy
visualether serve out --port 8080
```

In lazy mode, opening a session diagram triggers PDF generation for that session. The browser either receives the finished PDF directly or a temporary loading page while the PDF is being rendered.

```
# Multi-pcap parent directory: Capture Atlas tree view at /
visualether generate --fxt sessions.fxt.xml --input run-A.pcap \
  --output out/run-A --format lazy
visualether generate --fxt sessions.fxt.xml --input run-B.pcap \
  --output out/run-B --format lazy
visualether serve out --port 8080   # tree view at /, lazy PDFs work for both
```

```
# See what's running, then free the port held by the MCP-auto-spawned
# serve without exiting MCP (leaves serves you started yourself alone)
visualether serve list
visualether serve stop
```

```
# Anchor a whole workspace, then stop everything when done
visualether serve youtube-main          # one landing page for every capture
below
visualether serve stop --all            # stop the anchor and the auto-spawned
serve
```

`visualether serve` is the easiest way to serve VisualEther output, especially for lazy mode.

18.6.3 Using a Different Web Server

`visualether serve` is a convenience, but other web servers can also be used depending on the output mode:

- **Professional / Server `generate` output**: Uses relative links and external data files. Any web server can serve these files with no special configuration — including static hosts like GitHub Pages, S3, or a basic nginx setup. Most production web servers (nginx, Apache, Caddy) and CDNs compress responses on the fly, so clients still receive compressed data.
- **`--format lazy`**: Use `visualether serve` when you want on-demand PDF rendering. The built-in server watches for PDF requests, renders missing PDFs from the corresponding `.ndjson` files, and serves them once ready. A plain static web server cannot perform this on-demand rendering step on its own.
- **`--gzip`**: Output is gzip-compressed HTML and JSON, typically reducing disk usage substantially. This saves significant disk space when generating many session diagrams and eliminates runtime compression overhead on the server. Your server must send the `Content-Encoding: gzip` header for these files. For example, with nginx:

```
location /visualether/ {
    root /var/www;
```

```
    gzip_static on;
}
```

18.7 The `clean` Command

Remove the files VisualEther generated from an output directory while preserving your input files. Run it between format swaps so the Capture Atlas surfaces the diagram from your latest run instead of a leftover file from an earlier one.

```
visualether clean -o <DIR> [OPTIONS]
```

18.7.1 Options

Flag	Description
`-o, --output <DIR>`	Directory to clean. Falls back to `[clean].output` (or `[generate].output`) in `visualether.toml` when omitted.
`--dry-run`	Preview what would be removed without deleting anything.
`--force`	Skip the “looks like a VisualEther output” safety check.

18.7.2 What gets removed

Every diagram VisualEther rendered — PDF, HTML, NDJSON, and Markdown — together with the sidecar data files that back the interactive viewers. Empty subdirectories are pruned afterwards, and the target directory itself is always preserved. A `README.md` is kept (see below).

18.7.3 What gets preserved

Pattern	Reason
`*.pcap`, `*.pcapng`, `*.pcap.gz`, `*.pcapng.gz`	Input captures
`*.pdml`	PDML input
`*.fxt.xml`	FXT templates
`hosts.txt`	Host-to-IP mapping
`visualether.toml`	Project configuration
`README.md`	User documentation (case-insensitive)

Any unrecognized extension	Treated as user content

18.7.4 Safety guard

`clean` refuses to run unless the target directory actually contains VisualEther output, so it can't wipe a working directory that merely happens to share the configured output path. Pass `--force` to override.

18.7.5 Examples

```
# Clean the configured [clean].output (or [generate].output) — no flag needed
visualether clean

# Clean a specific directory
visualether clean -o output/

# Preview the deletion first
visualether clean -o output/ --dry-run

# Clean even if the directory doesn't have obvious VisualEther output
visualether clean -o some/dir --force
```

18.7.6 Typical workflow

```
visualether new bgp                          # creates bgp/ with seeded
visualether.toml
cd bgp
visualether generate                          # writes to output/ (per
[generate].output)
# … realize you want HTML output instead of combined
visualether clean                             # wipes output/ — inherits from
config
visualether generate --format html            # regenerates without leftover
PDFs
```

18.7.7 TOML-to-CLI Mapping

The `[clean]` section in `visualether.toml` mirrors the CLI flags. CLI flags always win; the config supplies defaults.

TOML key	CLI flag	Notes
`output`	`-o/--output`	Falls back to `[generate].output` when this section is absent.

`dry-run`	`--dry-run`	
`force`	`--force`	Use with care.

`visualether new` / `visualether init` seed a `[clean]` block matching `[generate].output`, so `visualether clean` works without arguments in freshly-created projects.

18.8 The `mcp` Command

Manage the MCP (Model Context Protocol) server for AI agent integration:

```
visualether mcp <SUBCOMMAND>
```

Subcommand	Description
`server`	Start MCP server (stdio transport).
`install`	Register VisualEther MCP server with Claude Code.
`config`	Print the MCP server config for manual registration with any MCP client.
`uninstall`	Remove VisualEther MCP server from Claude Code.

`mcp install` is the one-command path for Claude Code. The MCP server itself is a standard stdio server, so any MCP-capable client can use it. `mcp config` prints the launch command — with the absolute path to your VisualEther executable already filled in — plus a link to the per-client setup page:

```
visualether mcp config
```

For copy-paste configuration blocks for Cursor, Windsurf, VS Code, Codex CLI, Gemini CLI, Claude Desktop, and Zed, see eventhelix.com/visualether/mcp-clients (also covered in Section 14).

18.9 The `license` Command

Manage the VisualEther license:

```
visualether license <SUBCOMMAND>
```

Subcommand	Description
`show`	Display current license information.
`activate`	Activate a license file.
`deactivate`	Remove installed license (revert to Community).

`verify`	Verify a license file without installing.

18.10 The `eula` Command

Display the End User License Agreement for the active edition:

```
visualether eula [--professional | --server] [-o <FILE>]
```

With no flag, the command prints the EULA matching the currently active license: Professional licenses see the Professional EULA, Server licenses see the Server EULA, and Community / unlicensed installs see a short note plus the Professional EULA as a preview of what a paid license would grant.

Flag	Description
`--professional`	Print the Professional Edition EULA regardless of the active license.
`--server`	Print the Server Edition EULA regardless of the active license.
`-o, --output <FILE>`	Write the EULA Markdown to `<FILE>` instead of printing it. Useful for archiving alongside a purchase record or attaching to internal compliance documentation.

The command works offline — no network connection is required. The Server EULA describes the additional rights a Server license grants: installation on one CI build agent / shared server, plus three developer seats.

18.11 The `manual` Command

Open the VisualEther User Manual PDF:

```
visualether manual
```

This command locates the bundled `manual.pdf` in the installation's `documents` directory and opens it in the default PDF viewer.

18.12 Machine-Oriented Commands

The `fields`, `query-fields`, `endpoints`, and `protocols` commands produce JSON output designed for programmatic consumption. The MCP server and Claude Code call them to detect protocols, extract field metadata, query per-frame field values, and discover endpoints when generating FXT templates and hosts files.

Most users do not need these commands directly. To interactively explore fields in a PCAP, use `visualether analyze` instead — it produces a browsable HTML page where you can click to copy Wireshark field codes.

18.12.1 The `fields` Command

Extract all protocol fields from a capture file as JSON:

```
visualether fields [OPTIONS]
```

18.12.1.1 Options

Option	Description
`-i, --input <FILE>`	Input PCAP/PCAPNG file.
`-o, --output <FILE>`	Output JSON file.
`--protocols <PROTOCOLS>`	Filter fields by protocol prefixes (comma-separated). Example: `diameter,sip`.

18.12.1.2 Compact Mode

Option	Description
`--compact`	Output flat compact JSON format optimized for LLM token efficiency.
`--max-examples <N>`	Maximum showname examples per field in compact mode (default: 3).
`--exclude-empty <BOOL>`	Exclude fields with no examples in compact mode (default: `true`).
`--max-example-length <N>`	Maximum length for example values in compact mode (default: 80, 0 = unlimited).

18.12.1.3 Advanced Options

Option	Description
`--full-scan`	Scan the entire capture file instead of stopping early. By default, VisualEther stops scanning after 2000 consecutive packets with no new discoveries, which is fast and sufficient for most captures. Use `--full-scan` when working with diverse captures that contain many different protocols spread throughout the file.
`--tshark-args <ARGS>`	Additional arguments passed to tshark (shell-parsed string).
`--tshark-arg <VALUE>`	Additional tshark argument passed literally (repeatable).

18.12.2 The `query-fields` Command

Dump a **frame-indexed** `tshark -T fields` matrix — one row per frame, with the columns you choose — as a compact JSON envelope. Where `fields` answers *"what fields exist in this capture"* (values grouped by protocol), `query-fields` answers *"what is the value of these fields on each frame"*, which is what you need to check a cross-frame pattern: a HARQ process round-robin, a monotonically increasing sequence number, an SFN/slot or buffer-status timeline. This is primarily an MCP tool; the CLI form documented here exists for scripting and for reproducing what the MCP `query_fields` tool runs.

```
visualether query-fields --input <FILE> --fields <F1,F2,...> [OPTIONS]
```

Option	Description
`-i, --input <FILE>`	Input PCAP/PCAPNG file.
`-o, --output <FILE>`	Output JSON file. When omitted, the JSON envelope is printed to standard output.
`--fields <NAMES>`	Comma-separated Wireshark field names to extract, in column order (required). `frame.number` is prepended automatically as the first column — do not list it. Get exact names from `visualether fields` or the `analyze` field navigator.
`--display-filter <FILTER>`	tshark display filter (`-Y`), e.g. `"esp"` or `"mac-nr.harqid && mac-nr.direction==1"` . Scope the query with a filter rather than relying on the row cap so aggregate computations stay complete.
`--max-rows <N>`	Maximum rows to return (default: 2000). When the result is clipped, the envelope sets `truncated: true` . Aggregates over a truncated result are unreliable — narrow `--display-filter` instead of raising this for whole-capture computations.
`--tshark-args <ARGS>`	Additional arguments passed to tshark (shell-parsed string).
`--tshark-arg <VALUE>`	Additional tshark argument passed literally (repeatable).

The output is a thin envelope around a bare tab-separated `table` — the same shape `tshark -T fields` produces, so it stays compact:

Field	Meaning
`columns`	Column names in `table` order — always `frame.number` first, then the requested fields.
`row_count`	Number of rows (frames) in `table` .

`truncated`	`true` if the result hit `--max-rows`.
`decode_recipe_applied`	`true` when decode/framing tshark arguments were in effect (from a colocated `visualether.toml`, an auto-detected DLT_USER mapping, or `--tshark-arg` / `--tshark-args`). When `true`, an empty cell means the field is genuinely absent on that frame rather than undecoded.
`table`	The matrix as TSV: rows separated by newlines, cells by tabs. Multiple occurrences of a field within one frame are comma-joined (a frame carrying two ESP PDUs yields `0x0100,0x0100`).

Like every tshark-driven subcommand, `query-fields` inherits the project decode recipe from `visualether.toml` `[defaults]` (combined additively with any `--tshark-arg` / `--tshark-args` — see Section 18.13). So on a ciphered or DLT_USER capture that already has a `visualether.toml`, no decode flags are needed here: a field that only appears once the recipe is applied still resolves, and `decode_recipe_applied` reports `true`.

18.12.2.1 Example

```
# Per-frame ESP SPI + sequence (recipe inherited from the colocated
# visualether.toml; frame.number is added as the first column).
visualether query-fields --input capture.pcap \
  --fields esp.spi,esp.sequence --display-filter esp
```

18.12.3 The `endpoints` Command

Extract all unique IP addresses (or IP:port pairs) from a capture file as JSON. This command scans the entire capture without early-stop, ensuring complete endpoint discovery, including IPv6 addresses:

```
visualether endpoints [OPTIONS]
```

Option	Description
`-i, --input <FILE>`	Input PCAP/PCAPNG file.
`-o, --output <FILE>`	Output JSON file.
`--axis <LEVEL>`	Endpoint format: `ip` (default, unique IPs), `port` (unique IP:port pairs), or `mac` (link-layer endpoints — `eth.{src,dst}` and `wlan.{sa,da,ta,ra,bssid}`). Any other value is rejected with an error listing the accepted values.
`--tshark-args <ARGS>`	Additional arguments passed to tshark (shell-parsed string).

`--tshark-arg <VALUE>`	Additional tshark argument passed literally (repeatable).

The output JSON contains an `endpoints` array of sorted, unique addresses and a `packets_scanned` count. Use axis `ip` when generating hosts files; use axis `port` for detailed per-service analysis.

18.12.3.1 Limitation: DLT_USER captures (cellular radio)

`endpoints` extracts only layer-2/3 addresses (`ip.src` / `ipv6.src` / `ip.dst` / `ipv6.dst` for `--axis ip` and `--axis port` ; `eth.src` / `eth.dst` / `wlan.{sa,da,ta,ra,bssid}` for `--axis mac`). Cellular-radio captures recorded over `DLT_USER` (4G/5G MAC, NAS-EPS, S1AP, NGAP, etc.) carry the protocol bytes directly without an Ethernet or IP wrapper, so none of these fields exist in the capture, and `endpoints` returns an empty list.

This affects the bundled radio samples:

Sample	Encapsulation
`samples/4g-lte/`	DLT 148 (NAS-EPS), 149 (MAC-LTE), 150 (S1AP) — endpoints: 0
`samples/5g-nr-radio/`	DLT 149 (MAC-NR over synthetic UDP) — endpoints: 0
`samples/5g-srsran-ngap/`	DLT 152 (bare NGAP) — endpoints: 0

For comparison, real network captures with SCTP/IP-wrapped cellular signaling decode normally: e.g. `samples/4g-attach/` (S1AP/SCTP/IP) and `samples/5g-attach/` (NGAP/SCTP/IP) both return their participant IPs.

To extract participant identifiers from `DLT_USER` cellular captures, use the protocol-aware tools instead: `analyze` (field navigator) surfaces NAS / S1AP / NGAP identifiers such as IMSI, GUTI, RNTI, and gNB-ID; `generate` with the bundled session FXT renders per-UE / per-bearer sequence diagrams.

18.12.4 The `protocols` Command

Extract a list of protocols detected in a capture file as JSON:

```
visualether protocols [OPTIONS]
```

Option	Description
`-i, --input <FILE>`	Input PCAP/PCAPNG file.
`-o, --output <FILE>`	Output JSON file.
`--compact`	Output flat compact JSON format optimized for LLM token efficiency.
`--max-examples <N>`	Maximum showname examples per protocol (default: 3).
`--full-scan`	Scan the entire capture file instead of stopping early. By default, VisualEther stops scanning after 2000 consecutive

	packets with no new discoveries, which is fast and sufficient for most captures. Use `--full-scan` when working with diverse captures that contain many different protocols spread throughout the file.
`--tshark-args <ARGS>`	Additional arguments passed to tshark (shell-parsed string).
`--tshark-arg <VALUE>`	Additional tshark argument passed literally (repeatable).

18.13 Configuration File (`visualether.toml`)

VisualEther supports a project-level configuration file named `visualether.toml`. It is searched first in the directory of each `--input` (or positional pcap) path, walking up to the filesystem root, and then — if no match was found — in the current directory, walking up to the root. This means a project's `visualether.toml` is picked up automatically, even when you invoke `visualether` from an unrelated working directory, as long as the input path points into the project. When multiple inputs are anchored on different trees, the first input wins; the loaded path is printed on startup.

Precedence rule for single-value fields (e.g. `output`, `axis`, `fxt`, `hosts-file`, `format`, `monochrome`, …): command-line options override per-command section values, which override `[defaults]`. The first non-empty source wins.

Precedence rule for cumulative `tshark-arg` / `tshark-args`: the three sources are combined additively in the order `[defaults]` → per-command section → CLI. Repeated `-o` preferences resolve via `tshark`'s last-occurrence-wins semantics, so a CLI `--tshark-arg=-o --tshark-arg=foo:FALSE` still overrides a `[defaults] -o foo:TRUE` — without silently dropping the rest of the project's tshark args. This matches the way other tools handle repeated `-X` flags and avoids the common footgun where a single per-invocation flag shadows the entire project mapping.

Path resolution rule: relative paths inside `visualether.toml` are resolved against the directory the toml lives in, not the current working directory. This includes `output`, `hosts-file`, `fxt`, `input`, `fonts-dir`, the `root` for `[serve]`, and path-bearing tshark preferences (`kerberos.file:`, `tls.keylog_file:`). Relative paths on the command line stay CWD-relative as you would expect. Absolute paths in either location pass through unchanged. Together with the parent-search above, this means a project stays portable — copy the directory anywhere, and `output = "output/"`, `hosts-file = "hosts.txt"`, and the like keep resolving to siblings of the toml.

18.13.1 Supported Sections

- `[defaults]` — shared base inherited by every tshark-driven subcommand (`generate`, `fields`, `query-fields`, `protocols`, `analyze`, `endpoints`, `init`). Use it for `tshark-arg` / `tshark-args` that apply to every command in the project. Note that `query-fields` has no dedicated section — it reads its decode recipe from `[defaults]` only.
- `[generate]` — defaults for the `generate` command
- `[fields]` — defaults for the `fields` command
- `[protocols]` — defaults for the `protocols` command
- `[init]` — defaults for the `new` and `init` commands

- `[analyze]` — defaults for the `analyze` command (field navigator)
- `[endpoints]` — defaults for the `endpoints` command
- `[serve]` — defaults for the `serve` command (port, bind, idle-timeout)
- `[clean]` — defaults for the `clean` command

Cumulative semantics for `tshark-arg` / `tshark-args`: the three sources combine additively in order `[defaults]` → per-command section → CLI. tshark's last-`-o`-wins behaviour resolves per-pref conflicts naturally, so a CLI `--tshark-arg` can still override a `[defaults]` setting without dropping the rest of the project's tshark configuration.

`visualether new` (and `init`) writes the sample's `tshark-arg` into a single `[defaults]` block when the selected sample needs DLT_USER UAT mappings, heuristic dissectors, or `null_decipher` (for example, the cellular-radio and 5G samples). Every tshark-driven subcommand picks the args up automatically — no need to repeat the array under `[generate]`, `[analyze]`, and `[endpoints]`.

The generated `[generate]` section deliberately sets neither `fxt` nor `input`. `generate` falls back to:

- **FXTs**: auto-iterate over `explore.fxt.xml` and `sessions.fxt.xml` in the project directory (whichever exist), running once per file. Professional/Server Edition renders both diagrams from a single command; Community Edition skips `sessions.fxt.xml` (no session tracking) and renders only the explore PDF.
- **Inputs**: auto-discover `*.pcap`, `*.pcapng`, `*.pcap.gz`, and `*.pcapng.gz` in the project directory.

Set either explicitly only when you want a deviation — a custom FXT filename, or a subset of captures. CLI flags always win over the toml.

18.13.2 Example Configuration

```
# Shared dissector configuration — inherited by every tshark-driven
# subcommand (generate, fields, query-fields, protocols, analyze, endpoints,
init).
# Common case for cellular-radio captures: one DLT_USER UAT mapping,
# the matching heuristic dissectors, and (for null-cipher test rigs)
# the nas-5gs.null_decipher preference.
[defaults]
tshark-arg = [
    "-o",
    'uat:user_dlts:"User 2 (DLT=149)","udp","","","",""',
    "--enable-heuristic",
    "mac_nr_udp",
    "--enable-heuristic",
    "rlc_nr_udp",
    "--enable-heuristic",
    "pdcp_nr_udp",
    "-o",
    "nas-5gs.null_decipher:TRUE",
]
```

```
[generate]
# fxt omitted — generate auto-iterates over explore.fxt.xml and
# sessions.fxt.xml in this dir. Set explicitly to use a different name:
# fxt = "bgp_sessions.fxt.xml"
# input omitted — generate auto-discovers *.pcap[ng][.gz] in this dir.
# Set explicitly when you want a subset or specific filenames:
# input = ["bgp.pcap"]
output = "out"

# Rendering and output
format = ["combined"]
axis = "ip"
monochrome = false
timestamp-format = "absolute"

# Optional integrations
hosts-file = "hosts.txt"
fonts-dir = "fonts"
# tshark-arg omitted — inherited from [defaults] above. Add a
# per-command tshark-arg here only when generate needs args that
# the other subcommands should NOT see.

[fields]
input = "bgp.pcap"
output = "fields.json"
# tshark-arg inherited from [defaults]

[endpoints]
axis = "mac"
# tshark-arg inherited from [defaults]

[serve]
port = 8080
bind = "0.0.0.0"
idle-timeout = 600
no-open = true
```

18.13.3 TOML-to-CLI Mapping

The `[generate]` section uses the same option names as the `generate` command, with kebab-case keys where needed. For example:

TOML Key	CLI Flag

`fxt`	`--fxt`
`input = ["a.pcap", "b.pcap"]`	`--input a.pcap --input b.pcap`
`output`	`--output`
`format = ["pdf", "markdown"]`	`--format pdf --format markdown`
`output-mode = "json"`	`--output-mode json`
`axis`	`--axis`
`monochrome = true`	`--monochrome`
`gzip = true`	`--gzip`
`serve = true`	`--serve` (Professional / Server only)
`footer = "EventHelix — page {page}"`	`--footer "..."` (Professional / Server only)
`timestamp-format`	`--timestamp-format`
`hosts-file`	`--hosts-file`

18.14 Environment Variables

VisualEther reads the following environment variables for settings that are typically environment-specific rather than project-specific.

Variable	Description
`VISUALETHER_TSHARK_PATH`	Absolute path to the `tshark` binary. Use this when tshark is not on your `PATH` (e.g., a custom Wireshark installation).
`VISUALETHER_HOSTS_FILE`	Path to a hosts file for IP-to-hostname substitution in sequence diagrams.
`VISUALETHER_FONTS_DIR`	Path to a directory containing font files for non-Latin script rendering (Devanagari, CJK, Arabic, Hebrew) in PDF output.
`VISUALETHER_MAX_AXIS`	Hard cap on the number of axes (lifelines) in a single diagram, applied to both PDF and standalone HTML output. Diagrams beyond this point are unrenderable in practice — many PDF viewers fail or hang on extremely wide diagrams, and the HTML grid is too wide to read or scroll. Default: `64`. Applies to both Community Edition and Professional/Server Edition; the effective cap is the lower of this value and the edition's licensing cap (Community: 10).

`VISUALETHER_LICENSE_FILE`	Path to a license file. Useful in CI/CD pipelines or containers where the license is at a non-standard location.

18.14.1 `VISUALETHER_TSHARK_PATH`

If `tshark` is not on your system `PATH`, set this variable to the full path of the `tshark` executable:

```
# Linux / macOS
export VISUALETHER_TSHARK_PATH=/opt/wireshark/bin/tshark

# Windows (PowerShell)
$env:VISUALETHER_TSHARK_PATH = "C:\Program Files\Wireshark\tshark.exe"
```

When this variable is not set, VisualEther looks for `tshark` on the system `PATH`.

18.14.2 `VISUALETHER_HOSTS_FILE`

Set this variable to the path of a hosts file that maps IP addresses to friendly names:

```
export VISUALETHER_HOSTS_FILE=/home/user/project/hosts.txt
```

When this variable is not set, VisualEther automatically looks for a file named `hosts.txt` in the current working directory. If neither is found, no hostname substitution is applied.

18.14.3 `VISUALETHER_FONTS_DIR`

Set this variable to a directory containing font files (`.ttf`, `.otf`) for rendering non-Latin scripts such as Devanagari, CJK, Arabic, or Hebrew in PDF output:

```
export VISUALETHER_FONTS_DIR=/home/user/fonts/noto
```

This is especially useful for MCP server integration, where the subprocess inherits the parent's environment. Configure it once, and all MCP-generated PDFs will use the specified fonts without requiring `--fonts-dir` on every invocation.

18.14.4 `VISUALETHER_MAX_AXIS`

Set this variable to override the default cap of 64 axes per diagram. The cap applies to both PDF and standalone HTML output:

```
# Linux / macOS
export VISUALETHER_MAX_AXIS=96

# Windows (PowerShell)
$env:VISUALETHER_MAX_AXIS = "96"
```

The cap prevents output from becoming unrenderable with large numbers of axes — extremely wide PDFs crash many viewers, and an over-wide HTML grid is too wide to read or scroll.

When the cap is exceeded, VisualEther refuses to render that diagram and returns an error message listing three remediations:

- **Coalesce addresses with `hosts.txt`** (recommended) — map related addresses to the same friendly name. Multiple addresses sharing one name render as a single axis. Use this when many addresses represent one logical entity (an MLO AP advertised on three bands, IPv4/IPv6 of the same host, NAT'd peers, etc.).
- **Switch to sessions mode** — each session's diagram is naturally bounded to its participants, so the axis count per diagram stays low even on captures with thousands of distinct endpoints.
- **Raise the cap** — set `VISUALETHER_MAX_AXIS` to a higher value when you genuinely need wider diagrams.

Invalid or non-positive values default to 64.

18.14.5 `VISUALETHER_LICENSE_FILE`

Set this variable to point to a `license.lic` file when it is not in the standard search locations:

```
export VISUALETHER_LICENSE_FILE=/opt/licenses/visualether/license.lic
```

When this variable is not set, VisualEther searches for `license.lic` in the current directory (traversing up through parent directories), then in the platform-specific app data directory. Run `visualether license show` to see which file is being used.

18.14.6 Precedence

When multiple configuration sources specify the same setting, the most specific source wins:

Priority	Source	Example
1 (highest)	Command-line flag	`--hosts-file hosts.txt`
2	`visualether.toml`	`hosts-file = "hosts.txt"`
3	Environment variable	`VISUALETHER_HOSTS_FILE=hosts.txt`
4 (lowest)	Auto-detection	`hosts.txt` file in current directory

The same precedence applies to `fonts-dir`: `--fonts-dir` flag → `fonts-dir` TOML key → `VISUALETHER_FONTS_DIR` environment variable.

For `tshark`, VisualEther uses `VISUALETHER_TSHARK_PATH` when set, otherwise looks for `tshark` on your system `PATH`. There is no `--tshark` CLI flag — set the environment variable when you need to pin a non-default location.