VTX 1 Wire Specification: core

Oct 11, 2026, revised Nov 16, 2026 · VTX 1 · core · beta draft d03 · MIT licence · vtx.foundation/spec/v1/core/

This document specifies the core of VTX version 1 on the wire: transports, framing, channels, handshake, scene operations, input, resources and the trusted zone. It implements the decisions recorded in the VTX spec outline. Surfaces, audio, video, themes, redirection and the rest are extensions of VTX 1, each in its own document; this one defines the core every peer implements and the extension points the extensions use (Section 19 lists them).

§1 Status and conventions

This is the VTX 1 core specification, beta draft d03. The wire format may still change during the beta, so every transport identifier carries the draft number: ALPN `vtx1-d03` and `vtxq1-d03`, WebSocket subprotocol `vtx1-d03`, SSH subsystem `vtx-d03@vtx.foundation` and Noise prologue `vtx1-d03`. Peers built from different drafts therefore never connect and misread each other. Each extension carries its own draft number, which peers name in the `drafts` capability during the beta. When the core is frozen, the bare tokens (`vtx1`, `vtxq1`, `vtx@vtx.foundation`) take over; this document uses those names throughout. After the freeze, a breaking change requires a new major version and new tokens. Sections 17–19 define the application interface, fallback rendering and conformance.

Feature freeze. As of draft d02 (15 November 2026), the core is feature-complete. Draft d03 changes no frame or rule of the core: it moves the transport tokens to d03 and makes the extensions, until then grouped as v2 and v3, extensions of VTX 1, each with a conformance profile and a status (Section 19). Until the wire format freezes, the core takes only fixes that implementations, beta measurements or review turn up; new features go into extensions. The wire format freezes once independent implementations interoperate, beta deployments have measured flow control and the transports, an outside security review has been answered, and IANA has registered the port, ALPN IDs, WebSocket subprotocols and URI scheme. The bare tokens then replace the draft-numbered ones. Each extension freezes on its own, once it is stable.

Revision of 16 November 2026. Changes that keep every d03 peer compatible, so the transport tokens stay at d03: passkeys may be used from clients on other origins that the server lists (Section 5); integral JSON numbers outside the CBOR integer range stay floats in the text form (Section 7); READY may name the signed-in user for the trusted zone, an optional field that older peers ignore (Sections 5 and 12); a line of the application interface is an op when it has `op` (Section 17); and the fallback rendering of `status` and the prompt row on cell clients are defined (Sections 9 and 18).

The key words `MUST`, `MUST NOT`, `SHOULD`, `SHOULD NOT` and `MAY` are to be interpreted as described in BCP 14 (RFC 2119, RFC 8174) when, and only when, they appear in capitals.

Notation

- Byte layouts use the notation of RFC 9000 §1.3: `(i)` is a variable-length integer, `(..)` a variable number of bytes.

- Variable-length integers use the QUIC encoding (RFC 9000 §16). The two most significant bits of the first byte give a length of 1, 2, 4 or 8 bytes; values range up to 2^62 – 1.
- CBOR payloads (RFC 8949) are defined in CDDL (RFC 8610). Map keys are text strings in v1.
- JSON examples use the text form (Section 7).
- Resource hashes are SHA-256, written `sha256:` followed by 64 lowercase hex digits.

Terms

Term	Meaning
Client	The endpoint that renders and collects input
Server	The endpoint that hosts sessions and runs applications
Application	The program behind a session: a VT program, a VTX app or an AI agent
Session	Server-side state that survives individual connections
Channel	An ordered flow of frames inside a connection
Frame	The unit of transmission: channel, type, length, payload
Scene	The tree of nodes the client renders
Node	One element of the scene, identified by a string ID
Region	A layer 2 container with its own data model
Op	One scene operation, such as `open` or `cells`
Intent	A typed user action sent from client to application
Trusted zone	Client-drawn UI that no server content can draw into or imitate

§2 Transport bindings

VTX needs an encrypted, reliable, ordered byte stream. On QUIC-based transports each channel also gets its own stream, which removes head-of-line blocking between channels. Every binding carries the same frames (Section 3).

Binding	Carrier	Security	Channel mapping
QUIC	QUIC v1 (RFC 9000), ALPN `vtxq1`	TLS 1.3 (RFC 9001)	Channel 0 on client-initiated bidirectional stream 0; every other channel on one unidirectional stream per direction
WebTransport	WebTransport over HTTP/3; URI path selects the application	TLS 1.3	As QUIC, using WebTransport streams
TLS over TCP	TCP, ALPN `vtx1`	TLS 1.3	All channels multiplexed on one byte stream
WebSocket	RFC 6455 over HTTPS, subprotocol `vtx1`	TLS 1.3	As TLS over TCP; each binary message holds one or more whole frames
SSH	SSH subsystem `vtx@vtx.foundation`	SSH transport and user authentication	As TLS over TCP

Binding	Carrier	Security	Channel mapping
Serial	PPP over UART, USB CDC, radio links or pipes	TLS 1.3, as TLS over TCP; Noise in the experimental bare-metal profile	As TLS over TCP (below)

Rules for all bindings

- Every binding `MUST` encrypt the connection and authenticate the server. TLS-based bindings `MUST` use TLS 1.3 or later and refuse earlier versions. There is no unencrypted VTX.
- HELLO `MUST NOT` be sent as TLS early data (0-RTT), because early data can be replayed.
- On QUIC and WebTransport, a sender opens a channel's unidirectional stream when it first sends on that channel. All frames on one stream `MUST` carry the same channel ID.
- QUIC DATAGRAM frames (RFC 9221) are used only by extensions: live media (Media 1), forwarded UDP (Exec 1), viewer presence (Viewers 1), and gamepad state and sensor readings (Sensors 1).
- Browsers cannot query SVCB records, so for browser clients vtx://host[:port]/app maps to https://host[:port]/.well-known/vtx/app for WebTransport, and to wss:// at the same path for WebSocket. Browser clients use WebTransport where available and WebSocket otherwise.
- Over SSH, the SSH user is the VTX user: the server offers auth method `ssh` (Section 5). The subsystem name carries the `@vtx.foundation` suffix until IETF registration permits plain `vtx` (RFC 4250 §4.6.1).

Browser clients and origins. A server that accepts WebSocket or WebTransport connections `MUST` check the `Origin` header against an allowlist, and `MUST` refuse unauthenticated, `token` and `tls-cert` sessions from any origin outside it, so that a web page cannot drive a VTX server through a visitor's browser. The trusted zone's guarantees hold only when the server does not supply the client code (Section 12), so the reference web client is served from an origin run independently of any VTX server.

QUIC streams. Each endpoint `MUST` allow its peer at least 16 unidirectional streams and 1 bidirectional stream. RESET_STREAM or STOP_SENDING on the stream of channel 0–5 is a connection error `protocol_error`; on a dynamic channel it closes that channel.

Serial binding. In v1, a serial link carries PPP (RFC 1661, with RFC 1662 framing), and VTX runs over it with the TLS over TCP binding. Serial clients therefore get the same encryption and server authentication as any other. A bare-metal profile for devices without an IP stack (COBS-framed link packets with a CRC-32C, an ARQ layer, then Noise) stays experimental until the beta names one ARQ implementation, and is not required for conformance.

Noise. Where Noise is used, the pattern is `Noise_XX_25519_ChaChaPoly_BLAKE2s`, or `Noise_IK_25519_ChaChaPoly_BLAKE2s` when the client has the server's static key pinned. The prologue is the ASCII Noise token (`vtx1-d03` in this draft), and each Noise message is preceded by its length as a 2-byte big-endian integer. With XX, the client pins the server key on first use and refuses a changed key unless the user accepts it in the trusted zone. Noise proves a key, not a host name: the zone shows the key fingerprint as the server's identity, and the `webauthn`, `oidc` and `x509` sign-in methods, which all bind to a host name, are unavailable unless provisioning bound the key to one.

§3 Frame format

Every VTX message is a frame, and the frame layout is the same on every transport.

```
VTX Frame {
  Channel ID (i),
  Frame Type (i),
  Length (i),
  Payload (..),
}
```

Field	Meaning
Channel ID	The channel the frame belongs to (Section 4)
Frame Type	What the payload means; registered per channel (Section 16)
Length	Payload length in bytes
Payload	One CBOR data item, unless the frame type is defined as raw bytes

Rules

- A frame's payload MUST NOT exceed the receiver's `max_frame` limit (Section 5). Until WELCOME arrives, the limit is 4,096 bytes in both directions. No endpoint may advertise a `max_frame` below 1,024.
- CBOR payloads MUST use definite-length encoding and MUST contain exactly one data item. Receivers MUST ignore map keys they do not know, so extensions and later drafts can add fields without breaking older peers. A map with duplicate keys is invalid. Integral numbers MUST be encoded as CBOR integers and all other numbers as floats; NaN and infinities are not allowed, and a float where an integer is required is invalid.
- Receivers MUST skip frames of unknown type. A frame type that must not be skipped is only sent after the peer has accepted the capability that defines it.
- A truncated frame, an over-limit frame or an invalid CBOR payload is a connection error: `protocol_error` or `frame_too_large` (Section 14).

Frame types common to all channels

Type	Name	Payload (CDDL)	Use
0x01	CREDIT	`{ bytes: uint }`	Extends the peer's send window (Section 4)
0x02	CLOSE	`{ code: uint, ? reason: tstr }`	Closes a dynamic channel, or the connection on channel 0

Types 0x00 and 0x03–0x0F are reserved for future common frames.

Example. A CREDIT frame on channel 1 that grants 65,536 bytes is 15 bytes on the wire:

```
01                  Channel ID 1
01                  Frame Type 0x01 (CREDIT)
0C                  Length 12
A1                  CBOR map, 1 pair
   65 62 79 74 65 73   "bytes"
   1A 00 01 00 00      65536
```

§4 Channels and flow control

The well-known channels open implicitly when the server sends READY, so a session needs no extra round trips before it renders. Other channels open explicitly.

ID	Channel	Carries	Present
0	control	Handshake, authentication, channel management, keep-alive	Always, from connect
1	scene	Scene updates, acknowledgements, scene errors	Always, from READY
2	input	Keys, composition, pointer, wheel, focus, paste, intents	Always, from READY
3	resource	Content-addressed glyphs and fonts	From READY, when a `glyph.*` or `font` capability is accepted
4	zone	Approvals, permissions, claimed identity	Always, from READY
5	clipboard	Clipboard writes and reads	From READY, when `clipboard` is accepted
6–15	reserved	Future well-known channels	—
16 and up	dynamic	Surfaces, audio, video, file transfer (extensions)	Opened with OPEN

Dynamic channels. The client opens odd IDs and the server opens even IDs, so the two sides never collide. The opener sends OPEN on channel 0; the peer answers OPEN_OK or OPEN_REFUSED (Section 5). The opener `MUST NOT` send on the channel before OPEN_OK. Either side ends a dynamic channel with CLOSE on that channel. Well-known channels cannot be closed; a CLOSE on channels 1–5 is a `protocol_error`.

Flow control on stream bindings (TLS over TCP, WebSocket, SSH, serial):

- Every channel except 0 has a send window per direction. Its initial size is the receiver's `window` limit, 65,536 bytes by default, which `MUST` be at least the receiver's own max_frame.
- Sending a frame consumes its payload length from the window; CREDIT and CLOSE frames consume nothing. A sender may send a frame only when the window holds at least its payload length; otherwise it waits.
- The receiver returns window with CREDIT frames on the same channel as it consumes data. CREDIT adds to the window; a window above 2^62 – 1 is a flow_control_error.
- Channel 0 is not flow-controlled. Its frames `MUST` stay small and infrequent.

On QUIC and WebTransport, QUIC's stream flow control applies instead, and CREDIT frames `MUST NOT` be sent.

Ordering on QUIC. QUIC does not order streams relative to each other, so a frame can overtake READY. A client MUST buffer frames that arrive on channels 1–5 until it has processed READY, and frames on a dynamic channel until its OPEN_OK.

Channels after CAPS. When a CAPS answer newly accepts a `glyph.*` or `font` capability, or `clipboard`, channel 3 or 5 opens from the scene state the answer names. A capability that is dropped leaves its channel open but unused.

Priority. When several channels have data ready, senders SHOULD send in this order: control; input and zone; scene; resource; clipboard and dynamic channels.

§5 Connection establishment and capability negotiation

A session is ready one round trip after the secure transport is up, or two when the user signs in with a passkey or token.

> **Diagram: VTX handshake · discovery and TLS, then HELLO to READY.** Drawn in the PDF version, `01-vtx-1-wire-specification-core.pdf`.

Without sign-in, WELCOME and READY leave the server back to back, and the client renders after a single round trip.

1. **Discovery.** The client parses the URI (below). For `vtx://host/app` it queries SVCB at `_vtx.host`; for `vtx://host:port/app` it queries `_port._vtx.host` (Port Prefix Naming, RFC 9460 §2.3). Records without `alpn` are skipped. Without SVCB, the client tries SRV at `_vtx._udp.host` and `_vtx._tcp.host`, then A/AAAA on the default port, 22100 (Section 16). A host given as an IP literal skips DNS discovery.
2. **Connection racing.** When both `vtxq1` and `vtx1` are on offer, the client races them as Happy Eyeballs does (RFC 8305): QUIC first, TCP 250 ms later. The first handshake to complete wins.
3. **Secure transport.** TLS 1.3, an SSH session, or a Noise handshake (Section 2), with server authentication as below.
4. **HELLO** (client): versions, capabilities, limits and display.
5. **WELCOME** (server): the chosen version, the accepted capabilities and, if the user must sign in, the auth methods. A WebAuthn challenge and an OIDC nonce seed ride in WELCOME, so the client can answer at once.
6. **AUTH** (client), only when WELCOME lists methods. A failure returns AUTH_FAILED.
7. **READY** (server): session ID, resume token, scene state, `input_seq` and, optionally, the signed-in user. Channels 1, 2 and 4 are now open, and channels 3 and 5 when their capabilities were accepted.

When no sign-in is needed, the server sends WELCOME and READY back to back. The client renders every credential prompt in its trusted zone (Section 12), never in scene content.

URI. `vtx-URI = "vtx://" host [ ":" port ] path-abempty [ "?" query ] [ "#" fragment ]`, with the rules of RFC 3986; userinfo is not allowed. HELLO's `target` is the path and query exactly as written, percent-encoding kept, starting with `/` (an empty path becomes `/`). The fragment never leaves the client. Browser clients append the target to `/.well-known/vtx` (Section 2).

Server authentication. The client authenticates the origin host, which is the host in the URI, and never an SVCB or SRV target name; TLS SNI carries the origin host.

- **DANE.** When the TLSA RRset at `_port._udp.target` (QUIC) or `_port._tcp.target` (TCP) for the selected SVCB or SRV target is DNSSEC-secure, the client uses it as RFC 7671 describes, accepting the usages DANE-EE (3) and DANE-TA (2). The client validates DNSSEC itself, or relies on a validating resolver it reaches over an authenticated channel.
- A secure TLSA RRset that matches no certificate is fatal, and so is a bogus DNSSEC answer.
- With no TLSA records, or an insecure answer, the client validates the certificate for the origin host against WebPKI.
- The trusted zone shows "DANE" only after a TLSA match.

Handshake states and timeouts.

- Before WELCOME, the client sends only HELLO. Between WELCOME and READY, the only other frames allowed are AUTH, AUTH_CHALLENGE, AUTH_FAILED, PING, PONG and CLOSE; anything else is `protocol_error`.
- The server closes with `protocol_error` when no HELLO arrives within 10 seconds of the secure transport, and with `auth_failed` when sign-in has not completed 120 seconds after WELCOME, or after 5 failed attempts.
- After AUTH_FAILED with `retry: true`, the client may try any offered method. A `webauthn` retry uses a fresh challenge, which the server sends in AUTH_CHALLENGE; AUTH_CHALLENGE also serves methods with more than one step that are registered later.

Control frames (channel 0)

Type	Name	Direction	Payload
0x10	HELLO	C→S	Offer: versions, profile, caps, limits, display, locale, unicode, target, resume, client
0x11	WELCOME	S→C	version, unicode, accepted caps, server limits, server, auth offers, linger, idle
0x12	AUTH	C→S	method plus its credential
0x13	AUTH_CHALLENGE	S→C	method plus its next challenge
0x14	AUTH_FAILED	S→C	reason (credential, expired, revoked, unsupported or locked), message, whether retry is allowed
0x15	READY	S→C	session, token, state, resumed, input_seq, linger, user
0x16	OPEN	Both	channel, type, params
0x17	OPEN_OK	Both	channel, params
0x18	OPEN_REFUSED	Both	channel, code, reason
0x19	CAPS	Both	Client: a new full offer, optionally with new limits, locale and tz. Server: the full accepted set, answer: true and the scene state it applies from
0x1A	DISPLAY	C→S	New display metrics (resize, scale, dark mode, colours)
0x1B	PING	Both	nonce
0x1C	PONG	Both	The nonce echoed
0x1D	TOKEN	S→C	A fresh resume token

Type	Name	Direction	Payload
0x1E	DETACH	C→S	Keep the session; this connection is ending
0x1F	GOAWAY	S→C	code, reason, optional reconnect URI
0x20	SESSIONS	C→S	Empty; asks for the signed-in user's sessions, after READY
0x21	SESSION_LIST	S→C	sessions, each with session, app, title, attached

HELLO fields

Field	Type	Meaning
`versions`	array of uint	Supported major versions; this spec is 1. Draft builds are told apart by their beta identifiers (Section 16), not here
`profile`	0–3	Rendering profile, as a hint only; capabilities govern
`caps`	map	Offered capabilities (below)
`limits`	map	`max_frame`, `window`, `glyph_slots`, `cache_bytes`, `scene_nodes`, `retain`, `max_update`, `max_clip` (Section 14)
`display`	map	`cols`, `rows`, `colors`; optional `px`, `cell_px`, `scale`, `dark`, `contrast`, `reduced_motion`, and the colours `fg`, `bg` and `palette` (16 entries) that the VT engine reports to colour queries (Section 8)
`locale`	text	BCP 47 language tag
`tz`	text, optional	IANA time zone name, such as `Europe/Amsterdam`; datetime components use it (Section 9)
`unicode`	text	Unicode version for grapheme and width rules, e.g. `15.1`
`target`	text, optional	Application path and query from the URI
`resume`	map, optional	`session` and `state`, plus `token` when this device holds it; without a token, the user signs in as the session's owner (Section 6)
`client`	map, optional	`name`, `version`, `platform`
`inputs`	array of device descriptors, optional	The client's input devices (Section 11); a client may leave this out and send DEVICES after READY

HELLO `MUST` fit in 4,096 bytes; a client with many input devices may leave `inputs` out and send DEVICES after READY. WELCOME's `unicode` field states the version both sides then use: the lower of the client's and the server's, compared as major.minor numbers.

Capability negotiation

- The server's accepted set is a subset of the client's offer. Neither side may use a capability outside it.
- Unknown capability keys are ignored. Extension keys carry a domain, like `telemetry@example.com`.
- A value is `true` or a parameter, such as a list. The server answers with the parameter it chose.

- Only the client offers changes mid-session: CAPS with its full new offer, optionally with new `limits`, `locale` and `tz`, for example after it moves to a larger screen or the user changes language. The server answers with CAPS holding the full accepted set, `answer: true` and the scene state from which the new set applies. Before that state, the server removes or replaces nodes that depend on a dropped capability; until then the client renders their fallback.
- New limits, locale and time zone apply from that same state. Until then, the old limits hold, and the server must not exceed either the old or the new ones. The application hears of the change as a session event (Section 17).

Key	Value	Meaning
`cells`	true	Layer 1 grids (Section 8)
`glyph.bitmap1`	true	1-bit bitmap glyphs
`glyph.bitmap8`	true	8-bit coverage glyphs
`glyph.path`	true	Vector path glyphs
`font`	array of media types	Font resources: `font/woff2`, `font/otf`, `font/ttf`
`components`	array of catalog URIs	Layer 2 with these catalogs (Section 9)
`input.keys`	true	Distinct press, repeat and release key events
`input.ime`	true	Composition events
`input.pointer`	true	Pointer and wheel events
`input.touch`	true	Touch contacts
`clipboard`	array of media types	Clipboard channel; `text/plain;charset=utf-8` is required in the list
`zone.uv`	true	User-verified approvals, with WebAuthn or a smart card (Section 12)
`auth`	array of methods	Auth methods the client can perform
`input.pen`	`true`	Pen pressure, tilt, twist and hover
`input.lock`	`true`	Pointer lock with relative motion
`input.gestures`	array of gesture names	Gestures the client can recognise
`input.gamepad`	`true`	Gamepads and joysticks
`input.dictation`	`true`	Dictation from voice devices
`input.a11y`	`true`	Accessibility actions
`haptics`	array of effects	Haptic effects the client can drive

The extensions' keys are listed in `ext/registry.json` of the definitions bundle, each with its extension, and none repeats a key of the core. Keys that share a prefix form a family: `surface.` for surface kinds (`surface.vector`, `surface.image`, `surface.grid`, `surface.pixel`, `surface.a11y`, `surface.window`), `audio.` and `video.` for capture and clips, `input.` for input options (`input.keylock`, `input.datagrams`) and `redirect.` for what the device shares into a session (`redirect.drives`, `redirect.printers`, `redirect.media`, `redirect.smartcards`, `redirect.usb`). The others are `cells.sub`, `theme`, `composites`, `audio`, `video`, `transfer`, `exec`, `forward`, `notify`, `device`, `viewers`, `sensors`, `displays`, `broker` and `wasm`, and `drafts` during the beta.

Auth methods

Method	Flow
`tls-cert`	The client certificate already verified in TLS; AUTH carries no credential. Suited to devices: the certificate is requested during the TLS handshake, before the user has chosen how to sign in, and does not pass through load balancers that end TLS. People with smart cards use `x509`
`ssh`	The SSH-authenticated user; SSH binding only
`webauthn`	WELCOME carries a challenge of at least 16 random bytes; AUTH returns the passkey assertion (parameters below)
`oidc`	WELCOME names `issuer`, `client_id` and a 32-byte `nonce_seed`; the client signs in with authorization code and PKCE in the system browser, with the OIDC nonce derived from the seed and the server's host name (below), then sends the ID token. The server checks issuer, audience, azp, expiry and the nonce, so a token minted for one server cannot be relayed to another
`token`	A pre-provisioned bearer token for devices without a user, such as kiosks and IoT; sent only to the host it was provisioned for
`x509`	WELCOME carries a challenge; the client signs it with a certificate's private key, usually on a smart card, and AUTH returns the signature and the certificate chain (parameters below)

OIDC nonce and redirect. The nonce is `base64url(SHA-256("vtx1-oidc" || host || nonce_seed))` without padding, where `host` is the origin host the client authenticated, in lower-case ASCII (A-labels for internationalized names), and `nonce_seed` is the 32 bytes from WELCOME. This works the same over every binding, including browsers, which cannot reach a TLS exporter. Native clients use a loopback redirect, `http://127.0.0.1:{port}/vtx/oidc` with a port chosen per sign-in (RFC 8252 §7.3); browser clients use `/vtx/oidc` on their own origin. The provider must register these redirects for the `client_id`. Section 19 gives a test vector.

WebAuthn parameters.

- The rpId is the host name from the `vtx` URI, which the client has authenticated through TLS or SSH. Clients `MUST NOT` assert for any other rpId.
- `type` is `webauthn.get`. For sign-in the challenge is WELCOME's as given; for approvals it is defined in Section 12.
- `allowCredentials` is empty, so discoverable passkeys are used. `userVerification` is `preferred` for sign-in and `required` for approvals.
- Servers verify the rpId hash, the challenge and the signature. They accept the origin `https://` plus the rpId, every origin they list in `/.well-known/webauthn`, and the app origins of the native clients they accept.
- **Clients on another origin.** Browsers and native apps may use a passkey only for their own domain, or for a domain that vouches for them. A server that accepts browser clients from another origin, such as a web client on `app.example.net` for a server on `vtx.example.net`, `MUST` publish `/.well-known/webauthn` on its HTTPS host (WebAuthn related origin requests), listing those origins, for example `{"origins": ["https://app.example.net"]}`. It `SHOULD` publish `/.well-known/apple-app-site-association` (under `webcredentials`) and `/.well-known/assetlinks.json` for the native clients it accepts. The list is each server's own configuration, so no server depends on another domain.

- The rpId stays the host from the URI in every case: that binding keeps one server from passing off another server's sign-in as its own.
- A client that cannot use a passkey for the host, such as a browser without related-origin support or an app the host does not list, does not offer `webauthn`, and signs in with another method such as `oidc` or `token`. Approvals signed with a passkey (Section 12) follow the same rule.

x509 parameters. `x509` signs in with a certificate whose private key the client holds, usually on a smart card such as a PIV card, a national eID card or the Dutch UZI-pas, reached through PKCS#11, Windows CNG or macOS CryptoTokenKit.

- WELCOME's offer carries a 32-byte random `challenge`. It may list in `cas` the SHA-256 of the SubjectPublicKeyInfo of each CA the server accepts, so the client offers only certificates that chain to one of them.
- The client signs the bytes `"vtx1-x509" || host || challenge`, with `host` as for the OIDC nonce. AUTH carries `x509`: `certs` (DER, end-entity first, then intermediates), `alg` and `sig`, plus an optional stapled OCSP response in `ocsp`.
- `alg` is a COSE algorithm: ES256 (−7), ES384 (−35), ES512 (−36), EdDSA (−8), PS256 (−37), PS384 (−38) or RS256 (−257). ECDSA signatures are encoded as r ‖ s (RFC 9053 §2.1).
- The server validates the chain to a configured trust anchor (RFC 5280), requires the digitalSignature key usage, checks revocation by OCSP or CRL, verifies the signature and maps the certificate to a user. A revoked certificate fails with reason `revoked`, an expired one with `expired`.
- The PIN unlocks the card on the client. It is entered in the trusted zone, on the reader's own PIN pad or in the operating system's dialog, and never leaves the client.
- With `removal: "detach"` in the offer, the client clears the scene from the screen and sends DETACH when the card is removed; putting the card back resumes the session with a fresh sign-in (Section 6). The default is `none`.
- Web pages cannot reach a card's key, so browser clients offer `x509` only through an extension or a native helper. They can still use a card through `tls-cert`.

Section 19 gives a test vector.

Sign-in UI. While the user signs in, the trusted zone shows the server identity and, for `oidc`, the issuer; for `x509`, the certificate's subject and issuer.

§6 Session, state synchronization and resume

The server owns the scene; the client holds a numbered replica of it. A client that loses its connection resumes from the last state it applied, so a dropped link costs one catch-up, not a redraw.

Session. READY creates the session and gives it a 16-byte random ID. The session outlives connections: it ends when the application ends, or `linger` seconds after its last connection drops (no `linger` means 0). When the application ends, the server closes every connection with `session_ended` (Section 14), carrying the exit status or the signal when there is one; such a session cannot be resumed.

State numbers. Each scene UPDATE carries `state`, one higher than the previous update; the parts of a multi-frame update share one state (Section 7). READY reports the last state the server has produced: 0 for a new session, so its first UPDATE carries state 1.

Acknowledgement. The client sends ACK with the highest state it has applied. It SHOULD do so within 100 ms of applying an update and MAY cover several updates with one ACK.

Retention. The server keeps every update after the last acknowledged state, up to its `retain` limit in bytes. Beyond that it falls back to a snapshot.

Resume.

1. The client sends HELLO with `resume`: the session ID, the last state it applied and, when this device holds one, the resume token.
2. The server checks the token, when one is sent. Tokens are 32 random bytes and single-use. If WELCOME lists auth methods, the client signs in again, and the user MUST be the session's user; otherwise the resume fails. A resume without a token always requires sign-in.
3. On success the server sends READY with `resumed: true` and a new token. It then either replays the updates after the client's state up to READY's state, in order, or sends one UPDATE with `reset: true` and the state after READY's, holding the whole scene.
4. On failure the server starts a new session: READY carries `resumed: false`, and the client discards its old scene.

Pending requests. After a resume, the server sends again every zone and clipboard request that is still pending, with its original ID. A client that had already answered one sends the same answer again.

Input delivery. INTENT, PASTE and the commit of COMPOSE carry `seq`, which starts at 1 in each session and rises by one with each such event.

- READY's `input_seq` is the highest `seq` the server has delivered to the application; 0 for a new session.
- While connected, the server sends INPUT_ACK with that number at least once a second while any events are unacknowledged.
- The client keeps each such event until it is acknowledged. After a resume it sends again, in order, every one above READY's `input_seq`. The server drops a `seq` it has already delivered.
- Keys, pointer, wheel, gesture and gamepad events are never sent again: they belong to a moment that has passed, and the user sees the resumed screen before acting again.

Resuming from another device. A client without the token, such as a second device, sends HELLO with `resume` holding the session and state 0, and signs in as the session's user; the server then sends a reset snapshot, and the connection that held the session gets GOAWAY `taken_over` (below). After READY, a client may send SESSIONS to list the signed-in user's sessions; SESSION_LIST answers with each session's ID, application, title and whether a connection is attached.

GOAWAY. A client that follows a GOAWAY `reconnect` URI sends its resume token only when that URI names the same host; to any other host it resumes without the token and signs in again.

Token rotation. READY carries a fresh resume token, and the server sends TOKEN with another at least once an hour. The server keeps accepting the previous token until a resume has used the newer one, so a TOKEN lost with a dropped connection does not strand the client.

Detach. DETACH tells the server the client is leaving deliberately; its `linger` asks how long to keep the session. The server closes the connection at once with `no_error` and keeps the session for the requested time, capped at WELCOME's `linger`. WELCOME's `linger` is the server's maximum; READY's is the value in force.

One active connection in the core. A resume from a new connection takes the session over; the old connection receives GOAWAY with code `taken_over`. Several viewers with roles (owner, operate, interact, approve, watch) come with the Viewers 1 extension, which adds joining as a viewer alongside this resume.

Stale input. Every input event carries `seen`, the last state the client had applied when the event happened. Applications use it to recognise actions on UI that has since changed, such as a click on a removed button, and `MAY` ignore them.

Keep-alive. An endpoint sends PING after `idle`/3 seconds without sending anything. A connection that receives nothing for `idle` seconds is dead; the client then resumes. `idle` comes from WELCOME, defaults to 60 and `MUST` be at least 30.

Local prediction. Layer 2 fields echo typing locally (Section 9). A grid with `echo: "predict"` lets the client predict printable characters, as Mosh does. Predicted cells `SHOULD` be visually marked until an update confirms them.

§7 Scene model and operations

The scene is a tree of nodes, built and changed only through ops. Layer 1 and layer 2 share the same tree and the same op stream, so a terminal grid and an agent's form can sit side by side.

Nodes

- Every session starts with one node, `root`: a vertical stack that fills the display. The server can change its props with `patch`. Its `title` names the window or tab; the VT engine sets it from OSC 0 and 2.
- Each node has an ID, a type, a parent, ordered children and props.
- IDs are 1–64 characters from `A–Z a–z 0–9 _ . : -`, unique within the session. `root` is reserved.
- Every node accepts the common props `fallback` (plain text shown where the type is unsupported, and used by fallback rendering, Section 18), `label` (accessible name) and `hidden`, and the state and accessibility props of Section 9: `autofocus`, `disabled`, `busy`, `description`, `live`, `lang`, `dir` and `pointer_shape`.
- A node with `template: true` is not rendered itself; it is instantiated per item of a bound array (Section 9).

Ops

Op	Fields	Effect
`open`	`id`, `type`, `parent` (default `root`), `before`, props	Creates a node. If the ID exists with the same type, its props are replaced; with another type, it is an error
`patch`	`id`, `set`, `unset`	Sets or removes props
`append`	`id`, `text` or `lines`	Adds content to `text`, `code` and `log` nodes
`rows`	`id`, `rows`, `at`, `replace`	Inserts table rows at `at` (default: the end); `replace` swaps that many rows instead
`move`	`id`, `parent`, `before`	Re-parents or reorders a node
`close`	`id`	Marks a node complete; clients may drop loading indicators
`remove`	`id`	Deletes a node and its descendants
`data`	`region`, `path`, `value`	Sets a value in a region's data model at a JSON Pointer; without `value`, deletes it
`cells`, `fill`, `scroll`, `cursor`, `style`, `clear`	See Section 8	Layer 1 grid content
`glyph`	`slot`, `res`, `cells`	Binds a custom glyph slot to a glyph resource
`focus`	`id`	Moves keyboard focus to a node, unless the user moved focus in the last second (Section 9)
`attention`	`kind` (`bell` or `urgent`), `id`	Asks for the user's attention: a bell sound or flash, or an urgent mark on the window or tab. The optional `id` names the node it concerns
`begin`, `commit`	none	Text form only: group ops into one update

Rules

- The ops of one update apply in order and render atomically: the client never shows a half-applied update.
- A parent `MUST` exist before its children are opened.
- An op that fails is skipped. The client reports it with a scene ERROR and continues with the next op, so one bad line costs one node, not the screen.
- Clients check each op against the node types they know. Unknown props are ignored; a node of an unknown type renders its `fallback` and is reported as `unknown_type`; an unknown op is skipped as `invalid_op`; a bad value is skipped as `schema`; an unknown ID or missing parent is `unknown_id` or `parent_missing`.

Text in the scene. Strings in scene ops must not contain C0 control characters other than tab and line feed, nor DEL or C1 controls; an op with one is invalid (`schema`). Escape sequences reach a terminal only through the VT engine, which turns them into ops. In `code`, `diff` and grids, where they could hide what a command does, clients show bidirectional controls and zero-width characters visibly. Copying yields the semantic text, not its rendering.

Extension types. A type of the form `prefix:type` comes from a catalog that the enclosing region declares under that prefix (Section 9). Such a node must carry `fallback` and may carry props the core schema does not know. A client without that catalog renders the fallback; a prefix the region does not declare is `unknown_type`.

Scene frames (channel 1)

Type	Name	Direction	Payload
0x10	UPDATE	S→C	`{ state: uint, ops: [* op], ? reset: bool, ? more: bool }`
0x11	ACK	C→S	`{ state: uint }`
0x12	ERROR	C→S	`{ state, index, code, ? id, ? path, ? message }`

ERROR codes: `unknown_id`, `unknown_type`, `invalid_op`, `schema`, `parent_missing`, `limit`, `resource`. With `reset: true`, the client clears the scene back to an empty `root` before applying the ops.

Large updates. An update larger than `max_frame` is split over several UPDATE frames with the same `state`; every part but the last carries `more: true`. The client applies the whole update atomically when the last part arrives. Reset snapshots are sent the same way. All parts together must stay within the client's `max_update` (default 1 MiB), reset snapshots included, so the server keeps the whole scene within it; a larger update is `limit_exceeded`.

Text form. Applications and agents write ops as JSON Lines: UTF-8 JSON (RFC 8259), one op per line, media type `application/vnd.vtx.jsonl` (to be registered in the vendor tree; file extension `.vtxl`).

- Ops between `{"op":"begin"}` and `{"op":"commit"}` form one update. Outside a group each line is its own update, and the server MAY merge lines that arrive within one frame interval (about 16 ms).
- The server library adds `state` and turns each JSON op into a CBOR map with the same keys and values. JSON numbers with an integral value from −2^64 to 2^64 − 1 become CBOR integers, and all other numbers become floats (Section 3); JSON objects with duplicate names are rejected. Every value survives the round trip, so agents never handle CBOR and clients never parse JSON.
- `vtx-ops-1.schema.json` (Appendix) is the producer contract for the core catalog: every line an application or agent writes validates against it. Other catalogs publish their own schemas.

Applications. Applications produce these ops through the application interface (Section 17), which carries them as JSON Lines; the server turns them into UPDATE frames.

```
{"op":"begin"}
{"op":"open","id":"ops","type":"region","title":"Agent
fleet","catalog":"tag:vtx.foundation,2026:vtx/catalog/core/1"}
{"op":"open","id":"agents","type":"table","parent":"ops","cols":
[{"key":"name","label":"Agent"},{"key":"state","label":"State","kind":"status"}]}
{"op":"rows","id":"agents","rows":[["build-07","warning"],["index-02","ok"]]}
{"op":"close","id":"agents"}
{"op":"commit"}
```

On the wire this becomes one UPDATE frame on channel 1, shown in CBOR diagnostic notation:

```
{"state": 42, "ops": [
  {"op": "open", "id": "ops", "type": "region", ...},
  {"op": "open", "id": "agents", "type": "table", ...},
  {"op": "rows", "id": "agents", "rows": [["build-07", "warning"], ["index-02",
"ok"]]},
  {"op": "close", "id": "agents"}
]}
```

§8 Layer 1: grids, cells and glyphs

A `grid` node is a character-cell surface. It is what the VT engine renders legacy programs into and what the smallest clients implement. Every cell keeps its semantic text, even when it shows a custom glyph.

Grid props

Prop	Type	Meaning
`cols`, `rows`	uint	Size in cells; change with `patch`
`scrollback`	uint	Lines the client retains above row 0 (default 0)
`styles`	map of style ID → style	Named styles that runs refer to; style `0` is the grid default
`font`	text	A family name or a `sha256:` font resource (Section 10)
`keys`	`raw` or `none`	`raw` sends all keys to the server while the grid has focus (default `raw`)
`echo`	`none` or `predict`	Local prediction of typed characters (Section 6)
`mouse`	`none`, `click`, `drag` or `move`	Which pointer events the grid reports, as in xterm's mouse modes
`pointer`	`normal` or `lock`	`lock` captures the pointer for relative motion while the grid has focus, with the `pointer.lock` permission (Section 12)
`size`	`fixed` or `fill`	With `fill`, the client sizes the grid to the space it is given and reports the size in cells with RESIZE (Section 11); the server answers with a `patch` of `cols` and `rows`, and until then the client clips or pads. Default `fixed`

Style object. `fg`, `bg` and `ul_color` are colours; `ul` is `none`, `single`, `double`, `curly`, `dotted` or `dashed`; `bold`, `dim`, `italic`, `strike`, `blink` and `reverse` are booleans; `link` is an `https`, `http` or `mailto` URI that the user opens through the trusted zone (Section 12). Links with other schemes are ignored.

Colours are `#rrggbb`, a palette entry `p0`–`p255` (the xterm 256-colour palette), or a semantic role: `fg`, `bg`, `muted`, `accent`, `info`, `success`, `warning`, `critical`. A client with fewer colours maps them down; roles let it pick the closest meaning rather than the closest RGB.

Palette. Entries `p0`–`p15` and the roles `fg` and `bg` follow the client's theme, and the client reports their values in DISPLAY's `palette`, `fg` and `bg` (Section 5), so the VT engine can answer colour queries. Entries `p16`–`p255` are fixed, as in xterm.

Grid ops

Op	Fields	Effect
`cells`	`id`, `row`, `col`, `runs`, `erase`, `wrap`	Writes runs of text from (`row`, `col`) to the right; `erase: "eol"` clears the rest of the row to style `0`
`fill`	`id`, `rect` [row, col, rows, cols], `t`, `s`	Fills a rectangle with one grapheme (default a space) in one style
`scroll`	`id`, `n`, `top`, `bottom`	Scrolls rows `top`–`bottom` by `n`; positive moves content up. When `top` is 0, lines leaving the top enter scrollback
`cursor`	`id`, `row`, `col`, `shape`, `visible`, `blink`	Places the cursor; `shape` is `block`, `bar` or `underline`
`style`	`id`, `sid`, `style`	Adds or replaces a named style, and cells that use it change with it; without `style`, deletes it, and its cells fall back to style `0`
`clear`	`id`, `scrollback`	Clears every cell to a space in style `0`; with `scrollback: true`, drops the scrollback too

Runs. A run is `{ t, s, g }`:

- `t` is the text. Graphemes are extended grapheme clusters (UAX #29). East Asian Wide and Fullwidth characters (UAX #11) and emoji presentation sequences take two cells; all others take one. Both sides apply the negotiated Unicode version, so client and server always agree on placement.
- `s` is a style ID (default `0`) or an inline style object, for a one-off style that needs no entry in `styles`.
- `g`, optional, lists one glyph slot per grapheme of `t`. The client draws each slot's glyph and keeps `t` as the cell's text for copy, search, screen readers and fallback.
- Text past the last column is clipped. Grids never wrap by themselves; the VT engine wraps before it sends. A `cells` op with `wrap: true` marks its row as continuing on the next, so copy and search join the two rows and a client that reflows can rejoin them.

```
{"op":"cells","id":"term","row":2,"col":0,"runs":[{"t":"build-07 ","s":"1"},
{"t":"waiting","s":"warn"}],"erase":"eol"}
```

Glyph slots. Ordinary characters need no slot: the client draws them from `t` with its font. Custom slots start at 0x110000 (1,114,112), above every Unicode code point, and only custom slots appear in `g`. The `glyph` op binds a slot to a glyph resource, `{"op":"glyph","slot":1114112,"res":"sha256:…","cells":1}`, and without `res` unbinds it. A glyph with `cells: 2` is drawn only for a grapheme that takes two cells; otherwise the client draws `t`. Until the resource has arrived, or if the client lacks the glyph capability, the client also draws `t`. The client's `glyph_slots` limit caps how many slots may be bound at once.

The VT engine. A server-side VT engine runs a legacy program on a pseudo-terminal and turns its output into grid ops. It MUST:

- set `TERM=xterm-256color` and `COLORTERM=truecolor` for the program, and `LANG` from the session locale with UTF-8;
- interpret xterm-compatible output, including the alternate screen (as a second grid, toggled with `hidden`), scroll regions and OSC 8 hyperlinks;
- map the mouse modes 1000, 1002 and 1003 to the grid's `mouse` prop (`click`, `drag`, `move`), and encode reports as SGR (mode 1006) when the program asks for it;

- report focus changes from FOCUS events when the program enabled mode 1004;
- send everything between the start and end of synchronized output (mode 2026) as one UPDATE;
- map DECSCUSR to the `cursor` op's `shape` and `blink`;
- answer DA1, DA2 and XTVERSION itself, and the OSC 4, 10 and 11 colour queries from DISPLAY's `palette`, `fg` and `bg`;
- set the root `title` from OSC 0 and 2, send `attention` with `bell` for BEL, and set the grid's `pointer_shape` from OSC 22;
- turn OSC 52 writes into clipboard writes (Section 13), and answer OSC 52 reads only with the `clipboard.read` permission, with an empty answer otherwise;
- turn KEY events into the encoding the program asked for: legacy escape sequences, xterm modifyOtherKeys or the Kitty keyboard protocol;
- strip ESC, all other C0 controls except tab, carriage return and line feed, and all C1 controls from PASTE text, so a paste cannot end bracketed paste early or inject commands;
- deliver the cleaned text as bracketed paste when the program enabled it; if it did not and the text holds a line break, first ask the user to confirm through an APPROVAL;
- send RESIZE and DISPLAY size changes to the program as a window size change (SIGWINCH);
- drop Kitty graphics and sixel output, which Surfaces 1 maps to image surfaces.

§9 Layer 2: components, data model and intents

Applications describe what a component is; the client decides how it looks. The v1 catalog is `tag:vtx.foundation,2026:vtx/catalog/core/1` (a tag URI, RFC 4151). The client offers it in the `components` capability, and a server `MUST NOT` open component types outside the accepted catalogs.

Regions. A `region` node holds layer 2 content and its own data model. Every component sits inside a region, and a binding resolves against the nearest enclosing region. A region corresponds to an A2UI surface. Props:

- `title`;
- `catalog`: the URI of the catalog its plain type names come from; without it, the core catalog `tag:vtx.foundation,2026:vtx/catalog/core/1`;
- `catalogs`: a map from prefix to catalog URI, for types written `prefix:type` (Section 7);
- `send_data`: when `true`, intents carry the region's whole data model.

A server uses only catalogs the client accepted in its `components` capability. Catalog URIs are immutable: a changed catalog gets a new URI, and declares the one it builds on in `extends`, so a client that knows the older catalog still renders the types they share. The core catalog also lists the icon names that `icon` accepts.

Data model and bindings

- Each region has a JSON value, initially `{}`. The `data` op writes into it at a JSON Pointer (RFC 6901): a missing member is created, missing parents are created as objects, `-` as the last token appends to an array, and a write below a value that is neither object nor array is skipped as `schema`. Without `value` the member is removed; removing an array element shifts the later elements down.
- Props marked *dynamic* accept a literal, a binding `{"bind": "/agents/0/name"}`, or a formatter `{"fn": …}`.

- Input components bind two ways. The client writes user edits into the model at once, re-renders everything bound to it, and sends nothing until an intent fires.
- A container with `each: { path, template }` repeats its template node once per item of the array at `path`. Inside a template, a binding without a leading `/` is relative to the item.
- Template instances get IDs of the form `node#index`, one `#index` per template level, such as `page#1`. These IDs appear in input targets, intents and scene errors; the server addresses the template itself.

Core catalog

Type	Key props	Fallback rendering
`stack`	`dir` v/h, `gap` 0–4, `align`, `justify`, `wrap`, `each`	Children in order; horizontal children joined by two spaces
`split`	`dir`, `sizes`	Children stacked vertically
`group`	`title`, `style` plain/card	Title line, then children indented
`tabs`, `tab`	`selected`; tab: `title`	Tab titles on one line, the selected tab's content below
`divider`	`axis`	A line of ─
`overlay`	`modal`, `title`, `dismiss`	Framed block after the content
`text`	`text` (dynamic, inline Markdown), `role`, `size`	Plain text; links as `text <url>`
`heading`	`text`, `level` 1–3	Text, underlined at levels 1–2
`markdown`	`text` (dynamic): CommonMark with tables; no images, no raw HTML	The Markdown source, wrapped to width
`log`	`lines`, `max`, `follow`	Lines as-is
`code`, `diff`	`text`, `lang`	Verbatim
`kv`	`items` [[key, value]]	Aligned `key: value` lines
`table`	`cols`, `rows`, `key`, `sort`, `select`, `activate`, `each`	Aligned columns under a header, truncated to width
`tree`, `item`	item: `label`, `icon`, `open`	Indented with ├─ and └─
`list`	`each`	One `-` line per child
`progress`	`value` 0–1 or null, `label`	`[#####-----] 50% label`
`status`	`state` ok/info/warning/critical/pending, `text`	`[ OK ]`, `[INFO]`, `[WARN]`, `[CRIT]` or `[WAIT]` by state, then the text
`chart`	`kind` line/bar/spark, `x`, `series`	One sparkline per series with its range
`icon`	`name`	Its `label`, if any
`form`	`submit`, `cancel` (intent names)	Fields as prompts, one after another
`input`	`kind` text/password/number/email/multiline, `label`, `bind`, `placeholder`, `checks`	`label:` prompt

Type	Key props	Fallback rendering
`choice`	`label`, `options`, `multi`, `style` radio/dropdown/chips, `bind`, `checks`	Numbered options
`checkbox`	`label`, `bind`	`[x] label`, toggled by y/n
`range`	`label`, `min`, `max`, `step`, `bind`	Number prompt with its bounds
`datetime`	`label`, `mode` date/time/datetime, `bind`, `checks`	Prompt with an ISO 8601 hint
`action`	`label`, `intent`, `role` primary/secondary/danger, `context`	Numbered menu entry

Table columns are `{ key, label, kind, align }`; `kind` is `text`, `number`, `status`, `progress` or `spark`. A table's `key` names the column whose value identifies a row. A table's `bind` binds its selection two ways: as an array of key values when the table has a `key`, otherwise as an array of row indices in the order the rows were sent. An activated row's INTENT carries `key` as well as `row`. `role` on `text` is one of the semantic colour roles (Section 8). Inline Markdown in `text` means CommonMark emphasis, strong emphasis, code spans and links (`https`, `http`, `mailto`) only: no images, no raw HTML, no blocks; the `markdown` component takes blocks and tables too. Links in either open through the trusted zone (Section 12). Exact prop types are in `vtx-ops-1.schema.json`.

States and accessibility.

- `disabled` shows a control as unavailable and ignores input. `busy` shows a control or region at work and holds back its intents. `readonly` (on `input`, `choice`, `checkbox`, `range` and `datetime`) shows a value that can be copied but not changed. `error` shows a message with a control and marks it invalid.
- `label` is the accessible name; `description` adds a longer accessible description.
- `live` makes a node a live region: `polite` changes are announced when the user is idle, `assertive` ones at once. A `log` is a polite live region by default, and `status` and `progress` announce changes of state. Servers use `assertive` only for critical states.
- An `overlay` with `modal: true` is a dialog. A `chart` is exposed to assistive technology as a table of its data, and a grid as a terminal document.
- Clients map component roles to the platform's accessibility API: ARIA in browsers, UI Automation on Windows, NSAccessibility on macOS and iOS, AccessibilityNodeInfo on Android and AT-SPI on Linux.
- `lang` and `dir` set the language and direction of a node's text and its descendants; `pointer_shape` sets the pointer over a node.

Checks run on the client before submit, and the application `MUST` validate again: `required`; `regex` with `pattern`; `length` with `min`/`max`; `numeric` with `min`/`max`; `email`. Each may carry a `message`.

Formatters use the session locale: `formatString` (`template`, `args`), `formatNumber` (`value`, `decimals`), `formatCurrency` (`value`, `currency` as ISO 4217), `formatDate` (`value`, `style`) and `pluralize` (`count`, and a string per CLDR plural category: `zero`, `one`, `two`, `few`, `many`, `other`). The function names match A2UI's.

Exact semantics.

- Templates fill `{name}` placeholders from `args`, and `pluralize` strings fill `{count}`; `{{` and `}}` stand for literal braces.

- `pluralize` picks the category from the CLDR rules of the region's language (its `lang`, else the session locale). `other` is required and used for any category not given.
- `regex` uses I-Regexp (RFC 9485) and must match the whole value. `length` counts grapheme clusters. `email` accepts an RFC 5321 Mailbox.
- Number inputs and ranges write JSON numbers to the data model; an empty text field writes `""`.
- `datetime` values are RFC 3339 strings: full-date in `date` mode, partial-time in `time` mode, and a date-time with an offset in `datetime` mode. The client shows them in the time zone from HELLO or CAPS; `formatDate` takes the same strings.

Focus. Keyboard focus and Tab order are handled by the client, in document order. Initial focus goes to the first node with `autofocus`, else the first grid with `keys: raw`, else the first control. A modal overlay keeps focus inside itself until it closes. The client follows a `focus` op unless the user moved focus in the last second, and reports focus changes with FOCUS (Section 11).

Intents. When the user activates an `action`, submits a `form` or activates a table row, the client sends an INTENT on the input channel:

```
{"intent":"deploy","source":"deploy","region":"ops","values":
{"env":"production","note":"hotfix 2.4.1"},"seq":3,"seen":42}
```

Field	Meaning
`intent`	The intent name from the component
`source`	ID of the component that fired it. An action whose intent equals its form's submit or cancel acts for the form: the source is the form's ID, and a submit carries values. Inside a template, the instance ID
`region`	Its region
`context`	The action's `context`, with bindings resolved
`values`	For forms: each field's ID mapped to its value
`row`	For table activation: the row's index in the order the rows were sent, not as the client sorted them
`key`	For table activation, when the table has a `key`: the row's value in that column
`data`	The region's data model, when `send_data` is set
`seq`	Input sequence number, for delivery across resumes (Section 6)
`more`	`true` when the intent is too large for one frame and continues in the next INTENT with the same `seq`; the server joins the parts' `values` and `data` before delivering
`seen`	The last applied scene state (Section 6)

Applications receive intents in the same JSON form, validated by `vtx-intent-1.schema.json`. An agent therefore gets typed, validated input rather than raw events to interpret.

§10 Resources: glyphs and fonts

A resource is an immutable byte string named by its SHA-256 hash. Because the name is the content, a client fetches each glyph or font once and reuses it from cache in every later session with that server.

Resource frames (channel 3)

Type	Name	Direction	Payload
0x10	OFFER	S→C	`{ res, kind, size, ? meta }`
0x11	WANT	C→S	`{ res, ? offset }`; `offset` resumes a partial transfer
0x12	HAVE	C→S	`{ res }`
0x13	DATA	S→C	`{ res, offset, bytes }`
0x14	DROP	Both	`{ res }`: cancels a transfer, or the client refuses it

Transfer

1. The server sends OFFER for each resource the scene will use.
2. The client answers HAVE if the resource is cached, or WANT if not.
3. The server sends DATA chunks, each within `max_frame`.
4. The client checks the hash once all `size` bytes have arrived. On a mismatch it discards the bytes and sends WANT once more; after a second mismatch it treats the resource as unavailable.

Ops may refer to a resource before it arrives; the client renders the fallback until it does. A client over its `cache_bytes` budget answers DROP.

After READY. After every READY, new session or resumed, the server offers again each resource the scene refers to, since the client's cache may have dropped it. A client may send WANT for any hash the server has offered in this session.

Kinds

Kind	Content	Drawing
`glyph/bitmap1`	CBOR `{ w, h, bits }`, w and h 1–256: h × ceil(w/8) bytes, rows top-down, most significant bit first	Set bits in the run's foreground colour
`glyph/bitmap8`	CBOR `{ w, h, alpha }`, w and h 1–256: w × h coverage bytes	Foreground blended by coverage
`glyph/path`	CBOR `{ cmds }`: `["M",x,y]`, `["L",x,y]`, `["Q",x1,y1,x,y]`, `["C",x1,y1,x2,y2,x,y]`, `["Z"]`	Coordinates 0–4096 across the glyph's cell box, origin top-left, non-zero fill
`font/woff2`, `font/otf`, `font/ttf`	The font file	Used by `font` props

Bitmap glyphs are scaled to the cell box; clients SHOULD use whole-number scaling where they can. A glyph whose data length does not match `w` and `h` is unavailable. The hash of a glyph resource covers its CBOR bytes exactly as sent.

Caching and privacy. Clients SHOULD keep the cache across sessions. They MUST partition it by server identity, as browsers do, so that one server cannot probe for resources another server sent and use that to track the user.

Fonts. The application developer is responsible for the right to distribute the fonts it sends, exactly as with web fonts. Clients cache font resources the way browsers cache web fonts. Font parsers are a common attack surface, so clients MUST sanitise font files before use, as browsers do with OpenType Sanitizer, and MAY ignore TrueType hinting instructions.

§11 Input

Input travels on channel 2. The client declares its input devices, and every device event names its device, carries a timestamp and carries `seen` (Section 6). Device classes and event types are registries, so new kinds of input can join without a new protocol version.

Devices. HELLO's `inputs` lists the client's devices, and DEVICES reports devices added or removed later. A descriptor is `{ id, class, ? name, … }`; IDs are unique within a connection, and clients MAY coarsen `name` for privacy.

Class	Descriptor fields	Events
`keyboard`	`layout`: the platform's layout name, such as `de` or `us-intl`	KEY, COMPOSE
`mouse`	`buttons`; `trackpad`	POINTER, WHEEL; GESTURE from trackpads
`touch`	`contacts`: the maximum simultaneous contacts	POINTER, GESTURE
`pen`	`pressure`, `tilt`, `twist`, `hover` (booleans)	POINTER
`gamepad`	`mapping` (`standard` or `none`), `buttons`, `axes`, `haptics`	GAMEPAD
`joystick`	`buttons`, `axes`, `hats`, `haptics`	GAMEPAD
`voice`	`languages` (BCP 47) the client can dictate in	COMPOSE
`a11y`	none	A11Y
`name@domain`	Defined by the extension	EXT

Input frames (channel 2)

Type	Name	Direction	Payload
0x10	KEY	C→S	`{ dev, t, kind, code, key, ? text, mods, ? unshifted, ? target, seen }`
0x11	COMPOSE	C→S	`{ dev, t, phase, text, ? caret, ? confidence, ? seq, ? target, seen }`
0x12	POINTER	C→S	`{ dev, t, kind, type, ? id, x, y, ? dx, ? dy, ? col, ? row, buttons, ? button, ? contact, ? pressure, ? tilt, ? twist, ? w, ? h, ? samples, ? target, seen }`
0x13	WHEEL	C→S	`{ dev, t, dx, dy, unit, ? target, seen }`
0x14	FOCUS	C→S	`{ ? target, focused, seen }`
0x15	PASTE	C→S	`{ text, ? target, seq, ? more, seen }`
0x16	INTENT	C→S	Section 9
0x17	DEVICES	C→S	`{ ? added: [* device], ? removed: [* uint] }`

Type	Name	Direction	Payload
0x18	GESTURE	C→S	`{ dev, t, kind, phase, x, y, fingers, ? dx, ? dy, ? scale, ? rotation, ? vx, ? vy, ? direction, ? target, seen }`
0x19	GAMEPAD	C→S	`{ dev, t, ? buttons, ? axes, ? hats, ? target, seen }`
0x1A	HAPTIC	S→C	`{ dev, effect, ? strong, ? weak, ? left, ? right, duration }`
0x1B	A11Y	C→S	`{ dev, t, action, ? target, seen }`
0x1C	EXT	C→S	`{ dev, t, type, data, ? target, seen }`
0x1D	RESIZE	C→S	`{ target, cols, rows, ? px, seen }`
0x1E	INPUT_ACK	S→C	`{ seq }`

`t` is when the device produced the event, in microseconds on a monotonic client clock. Applications use it to measure latency, rebuild pen strokes and time gestures; it is never compared with the server's clock.

Keys

- `kind` is `press`, `repeat` or `release`. Clients without `input.keys` send `press` only.
- `code` is the physical key, using W3C UI Events `KeyboardEvent.code` values such as `KeyA`; left and right modifiers differ, as in `ShiftLeft`.
- `key` is the logical key, using UI Events key values such as `a`, `Enter` or `ArrowUp`.
- `text` is the text the key produced, if any.
- `mods` is a bitmask: 1 Shift, 2 Alt, 4 Control, 8 Meta, 16 Caps Lock, 32 Num Lock, 64 Hyper, 128 AltGraph.
- `unshifted`, optional, is the key value the key gives without Shift, such as `a` for `A`; the VT engine needs it for the Kitty keyboard protocol.

This removes the ambiguity of VT escape sequences, where Ctrl+I and Tab arrive as the same byte. Clients show their on-screen keyboard when a text field, or a grid with `keys: "raw"`, has focus and no hardware keyboard is present.

Routing

- Keys go to the focused node. Layer 2 fields handle text editing locally; the server sees the result in the next intent.
- A grid with `keys: "raw"` receives every key except those the client reserves.
- Every client `MUST` reserve at least one key combination that never reaches the server. It takes the user to the client's own controls and the trusted zone, much like SSH's `~.` escape, and always releases pointer lock.
- Gamepad state goes to the focused node only if it or an ancestor sets `gamepad: true`. Otherwise the client uses gamepads to navigate layer 2 itself, as game consoles do: the D-pad and left stick move focus, the south button activates and the east button goes back.

Composition and dictation. COMPOSE reports IME composition with `phase` `start`, `update`, `commit` or `cancel`. Layer 2 fields compose locally; grids receive the committed text, so CJK input works in legacy programs too. Dictation from a `voice` device uses the same events: interim results as `update`, the final result as `commit`, with

an optional `confidence` from 0 to 1. In the core, the user always starts dictation from the client's own controls, and the audio never leaves the client. Streaming voice audio to the server needs the audio channel of Media 1 and a `microphone` permission.

Pointer

- `kind` is `move`, `down`, `up`, `enter`, `leave` or `cancel`, plus `lock` and `unlock` when pointer lock starts or ends.
- `type` is `mouse`, `touch` or `pen`; touch and pen contacts carry an `id`.
- `x`, `y` are device-independent pixels from the display's top-left corner. Over a grid, `col` and `row` are added.
- `buttons` is the W3C bitmask of pressed buttons: 1 primary (pen tip), 2 secondary (pen barrel), 4 middle, 8 back, 16 forward, 32 eraser. `button` names the one that changed.
- Pens add `pressure` (0–1), `tilt` [x, y] and `twist` (0–359) in degrees, and `contact: false` while hovering in range. Touch contacts add their size as `w`, `h` in pixels.
- `samples` holds the high-rate samples coalesced since the previous event, oldest first, each `[t, x, y]` or `[t, x, y, pressure]`. Clients `SHOULD` send them for pens and `MAY` for mice, so strokes and fast motion are not lost to the frame rate.
- The client reports only what the target needs: a grid's `mouse` mode, or the pointer events the application asked for. Hover effects in layer 2 stay local.

Pointer lock. A node with `pointer: "lock"` asks for relative mouse input, as games and remote desktops need. On the user's first click in it, the client asks for the `pointer.lock` permission in the trusted zone, hides the cursor and sends a POINTER `lock`. While locked, move events carry the relative motion in `dx`, `dy`, and `x`, `y` stay fixed. The reserved key combination always releases the lock, and the client then sends `unlock`.

Gestures. A node lists the gestures it wants in its `gestures` prop: `tap`, `double-tap`, `long-press`, `pan`, `pinch`, `rotate`, `swipe`. The client recognises them with the platform's own recogniser, so they behave natively on every device, and sends GESTURE:

- `phase` is `start`, `update`, `end` or `cancel`; `tap`, `double-tap` and `long-press` report `end` only.
- `x`, `y` are the centroid; `fingers` is the number of contacts.
- `pan` reports `dx`, `dy` and `pinch` reports `scale`, both cumulative since `start`; `rotate` reports `rotation` in degrees; `swipe` reports `direction` (`up`, `down`, `left`, `right`). At `end`, `vx`, `vy` give the velocity in pixels per second.
- Contacts that form a recognised gesture are not also sent as raw touch events. Trackpad pinch and rotate arrive the same way, from the mouse device.
- Layer 2 components handle scrolling, zooming and taps locally, without any of this.

Gamepads and joysticks

- GAMEPAD carries the device's full current state. Clients send it when the state changes, at most 120 times a second per device.
- With `mapping: "standard"`, buttons and axes follow the W3C Gamepad standard mapping: 17 buttons and 4 axes. Other devices report them in device order.
- Button values run from 0 to 1, so analogue triggers keep their travel. Axes run from −1 to 1. A hat is −1 when centred, or 0–7 clockwise from up.

- With Sensors 1, GAMEPAD may travel as QUIC datagrams, where the newest state wins; Sensors 1 defines how, with every button change also sent on the stream.

Haptics. HAPTIC drives feedback on a device whose descriptor lists the effect: `rumble` with `strong` and `weak` (0–1), `trigger-rumble` with `left` and `right` (0–1), or `vibrate` on phones and tablets. `duration` is in milliseconds, at most 5,000. Clients MAY scale down or ignore effects, following the user's settings.

Accessibility actions. A11Y carries screen reader and switch-access commands: `activate`, `increment`, `decrement`, `scroll-up`, `scroll-down`, `scroll-left`, `scroll-right` and `dismiss`. Layer 2 components handle them locally; they reach the server for grids and, with the extensions, for surfaces, where Sensors 1 adds item roles, a pixel-surface overlay and the `focus` and `set` actions.

Wheel. `dx`, `dy` with `unit` `px`, `line` or `page`. Scrolling a layer 2 container is local; the wheel goes to the server only over a grid that reports the mouse, or when nothing local can scroll.

Paste. A user paste arrives as PASTE, never through the clipboard channel. A paste larger than `max_frame` is split: every part but the last carries `more: true`, and all parts share one `seq`. The server delivers the paste when its last part arrives and drops a partial paste when the connection drops; the client then sends every part again after the resume (Section 6). INTENT continues the same way. The VT engine cleans a paste and wraps it in bracketed paste (Section 8).

Resize. For a grid with `size: fill`, the client sends RESIZE with the cells that fit, and the size in pixels when known, whenever its space changes, at most once every 50 ms per grid while the user drags. The server answers with a `patch` of the grid's `cols` and `rows`, and the VT engine passes the size to the program.

Extensions. A device class outside the registry is named `name@domain`. The client offers it as a capability of the same name and sends EXT events only after the server accepts it; the extension defines `type` and `data`. Eye trackers, XR controllers or MIDI devices can join this way without a new protocol version.

§12 Trusted zone

Every client MUST reserve a trusted zone: UI that only the client draws, that scene content can never draw into, overlap, theme or imitate. It is what makes a VTX approval worth more than a button in a web page.

Requirements

- The zone always shows the server's identity (host and trust mode: WebPKI, DANE or pinned key), the connection state and the signed-in user. The client knows the user from its own sign-in; after `token` sign-in or a resume it takes READY's `user`, which it labels as reported by the server.
- It is visually distinct and stays reachable through the reserved key combination (Section 11).
- On very small displays the zone MAY be a mode the client switches into, provided no scene content renders while it is shown.
- Credential prompts (Section 5), smart-card PIN entry, approvals and permission requests appear only in the zone, or in the operating system's own dialogs.

Where the guarantee holds. The trusted zone's guarantees hold only where the server does not supply the client's code: installed clients, browser extensions, and a web client served from an origin that is independent of the server, such as the reference web client (Section 2). A web client that a VTX server serves itself runs code that server controls, and can draw a fake zone. A WebAuthn assertion names its origin in clientDataJSON; when that origin is the server's own, the assertion proves only that the user's authenticator was used, not what the user was shown. Applications record the origin with each approval, and may refuse high-risk approvals from the server's own origin.

Zone frames (channel 4)

Type	Name	Direction	Payload
0x10	APPROVAL	S→C	`{ id, title, ? detail, ? command, risk, requester, ? expires, ? uv, ? proof }`
0x11	APPROVAL_RESULT	C→S	`{ id, decision, digest, ? assertion, ? x509 }`
0x12	PERMISSION	S→C	`{ id, perm, ? detail, ? url }`
0x13	PERMISSION_RESULT	C→S	`{ id, decision, ? scope }`
0x14	CLAIM	S→C	`{ ? name, ? icon, ? color }`
0x15	CANCEL	S→C	`{ id }`: withdraws a pending request

Approvals

- `requester` is `{ name, kind }` with `kind` `agent`, `app` or `user`. `risk` is `low`, `medium` or `high`. `expires` is in seconds.
- The client shows every field the digest covers: title, requester, risk, server identity, and both `detail` and `command` when present. Text is shown verbatim: no Markdown, control characters made visible, bidirectional text isolated.
- For `risk: "high"` the client MUST require a deliberate act: no default button, and Enter alone does not answer.
- `decision` is `allow`, `deny`, `expired` or `unsupported`.
- `digest` is the SHA-256 of the APPROVAL payload bytes exactly as received. The server MUST check it, which binds the answer to the request the user actually saw. Approval IDs MUST be unique within a session, and a server accepts one answer per ID.
- With `uv: true`, the client proves that the user approved, using the first proof in `proof` it can give (default `webauthn`). Either proof covers the bytes `"vtx1-approval" || session ID || digest`, with the 16-byte session ID, which ties it to this session and this request. The application can store it as verifiable proof that this user approved exactly this request.
- `webauthn`: a WebAuthn assertion with the Section 5 parameters and user verification required, whose challenge is the SHA-256 of those bytes. The client returns `assertion`: `credential_id`, `authenticator_data`, `client_data_json`, `signature`.
- `x509`: a smart-card signature over those bytes, returned in `x509` with the fields of an `x509` sign-in (Section 5). The server checks the certificate as at sign-in, and it must map to the signed-in user. Cards usually hold a separate signing key that asks for the PIN on every use; that is the key to use.
- A client that can give none of the listed proofs, for example without `zone.uv` or without a passkey or card for the signed-in user, answers `unsupported`.

Limits. At most 8 zone and clipboard requests may be pending in a session. The server must not send a ninth; a client that receives one closes the connection with `policy_violation`. A client may deny a permission request at once when the user denied the same permission earlier in the session.

Permissions in the core: `clipboard.read`, `clipboard.write` (writes outside a user activation, Section 13), `open_url` and `pointer.lock` (Section 11). For `open_url` the zone shows the full URL, and only `https`, `http` and `mailto` URLs are allowed. `decision` is `allow` or `deny`; `scope` is `once`, `session` or `always`. Grants with scope `always` are stored per server identity, and the user can revoke them in the client. The extensions add `microphone`, `camera`, `screen` and `keyboard.lock` (Media 1), `clipboard.files` (Transfer 1), `recording` (Viewers 1), `motion`, `location`, `location.precise` and `environment` (Sensors 1), `fullscreen` (Windows 1), `drives` (Drives 1), `printers` (Printers 1) and `smartcards` (Smart Cards 1).

Claimed identity. CLAIM lets an agent present its name, an icon (a glyph resource, Section 10) and an accent colour as `#rrggbb`. The client shows it labelled as claimed by the server, visibly separate from the verified server identity, and ignores the icon if it lacks the glyph capability. This is where A2UI's `agentDisplayName` and `iconUrl` land.

§13 Clipboard

The clipboard channel lets an application copy to and, with permission, read from the user's clipboard. User pastes do not use it; they arrive as PASTE input (Section 11).

Type	Name	Direction	Payload
0x10	WRITE	S→C	`{ items, ? more }`
0x11	READ	S→C	`{ id, mimes }`
0x12	DATA	C→S	`{ id, items, ? more }`
0x13	DENIED	C→S	`{ id }`

An item is `{ mime, data }` with `data` as bytes.

- **Writes.** The client applies a WRITE without asking only within 1 second of a user activation: an intent the user triggered, a key press or a pointer down. Pointer moves, releases and key repeats do not count, and one activation allows one write. Otherwise the client asks for `clipboard.write` permission first. This blocks silent clipboard replacement.
- **Reads.** A READ always requires `clipboard.read` permission. If the user declines, the client answers DENIED.
- **Media types.** Only types accepted in the `clipboard` capability are used. `text/plain;charset=utf-8` is always supported.
- **Large items.** An item larger than `max_frame` is split. The first frame carries the item with the first part of its data; each following frame carries one item with the same `mime`, whose data is appended. Every frame but the last carries `more: true`.

Size. No clipboard item may exceed the client's `max_clip` (default 1 MiB), in either direction. The server sends no larger item. A client answers a READ with only the items that fit, or DENIED when none do.

§14 Errors and close codes

There are three levels of error, and each is contained at the smallest level that works.

- **Scene errors** affect one op. The client skips the op, reports it with a scene ERROR (Section 7) and keeps going.
- **Channel errors** close one dynamic channel with CLOSE on that channel. An error on a well-known channel is a connection error.
- **Connection errors** end the connection: the endpoint sends CLOSE on channel 0 with a code, then closes the transport. The client then resumes if the code allows it.

Code	Name	Meaning	Client resumes?
0x00	`no_error`	Normal close, including after DETACH	Only if the user reconnects
0x01	`protocol_error`	Malformed frame or a rule of this spec broken	No
0x02	`internal_error`	Implementation failure	Yes
0x03	`flow_control_error`	Sent beyond the window	No
0x04	`frame_too_large`	Payload above `max_frame`	No
0x05	`version_unsupported`	No common major version	No
0x06	`auth_failed`	Authentication failed and no retry is allowed	No
0x07	`taken_over`	The session resumed on another connection	No
0x08	`going_away`	Server shutting down or moving; see GOAWAY's `reconnect`	Yes
0x09	`idle_timeout`	Nothing received for `idle` seconds	Yes
0x0A	`limit_exceeded`	A negotiated limit was exceeded	No
0x0B	`refused`	Channel open refused (OPEN_REFUSED only)	Not applicable
0x0C	`policy_violation`	Zone or permission misuse, such as an approval flood	No
0x0D	`session_ended`	The application ended; CLOSE carries `exit` or `signal` when known	No

Codes 0x0E–0x3F are reserved for this specification and its extensions. The extensions assign 0x0E `user_stopped` (Media 1), 0x0F `revoked` (Devices 1), and 0x10 `not_admitted` and 0x11 `removed` (Viewers 1). 0x40 and up are free for implementations.

Limits and their errors

Limit	Exceeded by	Result
`max_frame`, and 4,096 bytes for HELLO	A frame payload	Connection error `frame_too_large`
`window`	Data sent beyond the credit	Connection error `flow_control_error`
`scene_nodes`, `glyph_slots`	An op that would exceed it	Op skipped, scene ERROR `limit`

Limit	Exceeded by	Result
`cache_bytes`	An offered resource	Client answers DROP
`retain`	Updates the client has not acknowledged	Server resumes with a reset snapshot
`max_update`	The parts of one update together	Connection error `limit_exceeded`
`max_clip`	A clipboard item	Server: connection error `limit_exceeded`; client: the item is left out of DATA
8 pending zone and clipboard requests	A ninth request	Connection error `policy_violation`
Any other negotiated limit	The peer	Connection error `limit_exceeded`

§15 Security considerations

VTX's security rests on three structural choices: no in-band control, no code on the client, and a trusted zone the server cannot reach. The rules below keep those properties intact.

Threat	Mitigation
Content that reprograms the client (the classic `cat` of a hostile file)	Control exists only as frame types; text in ops and cells is never interpreted as commands
Hostile server exhausting a client	Client limits: `max_frame`, windows, `scene_nodes`, `glyph_slots`, `cache_bytes`, `max_update`, `max_clip`, and at most 8 pending zone and clipboard requests; Section 14 lists the error for each
Content triggering network access	Clients never fetch on behalf of content; links open only through the `open_url` permission
Spoofed system UI	The trusted zone cannot be drawn into or themed; CLAIM is labelled as claimed
Approval answered for a different request	The answer carries the digest of the request as shown; the server checks it
Repudiated approval	`uv` approvals carry a WebAuthn assertion or a smart-card signature over the request digest
Server reusing an approval answer	Approval IDs `MUST` be unique within a session; servers reject a second answer to the same ID
Deceptive command text (bidirectional overrides, invisible characters)	Zone text shows control characters and isolates bidirectional runs
Clipboard hijacking	Silent writes only right after user input; reads always need permission
Cross-server tracking through the cache	The resource cache is partitioned per server identity
Stolen resume token	256-bit, single-use, rotated, only ever sent inside the encrypted session
Lost or stolen smart card	The card's PIN, revocation checked at every sign-in, and the `removal` policy that detaches the session when the card is pulled (Section 5)

Threat	Mitigation
Downgrade	TLS 1.3 only; a mismatched ALPN aborts; HELLO never in 0-RTT
Replayed input on changed UI	`seen` lets the application discard actions on stale state
A web page posing as a VTX client	Servers check the `Origin` of browser connections (Section 2); zone guarantees hold only for clients whose code the server does not supply (Section 12)
Escape sequences hidden in scene text	Control characters make a scene op invalid; only the VT engine turns terminal output into ops (Section 7)

Further protections defined elsewhere in this specification:

- **Token relay.** OIDC ID tokens are bound to the server's host and a per-connection seed through the nonce, and passkey assertions to the URI's host through the rpId (Section 5).
- **Ambiguous payloads.** Duplicate map keys make a payload invalid, so every decoder reads the same request (Section 3).
- **Paste injection.** The VT engine strips control characters from pastes before delivering them (Section 8).
- **Font parsers.** Font files are sanitised before use (Section 10).
- **Links.** Only `https`, `http` and `mailto` links open, always through the trusted zone (Sections 8, 9 and 12).
- **Unauthenticated links.** Noise XX pins the server key on first use and shows its fingerprint (Section 2).

Agents. VTX guarantees that an approval was really given by the user, for exactly the request shown. Deciding which actions need approval, and enforcing that, remains the application's job.

Privacy. HELLO reveals display metrics, locale, time zone, input devices and client software, and input timestamps reveal typing rhythm. Clients `MAY` coarsen these values when the user asks for less fingerprinting, for example by rounding pixel sizes, rounding `t` to 10 ms or more, and leaving out device names. Frame sizes and timing can reveal keystrokes to an observer of the network path; a client that cares pads records (TLS 1.3 record padding, QUIC PADDING frames) and sends keys on a fixed tick.

§16 Identifiers and registries

ScaiLabs maintains these registries during the beta. They move to the foundation or consortium when it forms. The specification and reference implementation are MIT-licensed.

Identifiers

Identifier	Value	Status
URI scheme	`vtx`	Provisional registration to request (RFC 7595)
TLS ALPN	`vtx1` (TLS over TCP), `vtxq1` (QUIC); in draft d03 `vtx1-d03`, `vtxq1-d03`	To request from IANA at freeze
SVCB service	`_vtx`, with Port Prefix Naming	Scheme mapping in Section 5; `_vtx` to enter the underscored node names registry (RFC 8552)
SRV services	`_vtx._udp`, `_vtx._tcp`	To request from IANA

Identifier	Value	Status
Default port (UDP and TCP)	22100 (0x5654, "VT" in ASCII), requested	Requested from IANA; beta implementations use it from draft d01 and move if IANA assigns another number. Deployments SHOULD still publish SVCB or SRV records, which take precedence
WebSocket subprotocol	`vtx1`; in draft d03 `vtx1-d03`	To request from IANA at freeze
WebSocket subprotocol, application interface	`vtx-app1`; in draft d03 `vtx-app1-d03`	To request from IANA at freeze
SSH subsystem	`vtx@vtx.foundation`; in draft d03 `vtx-d03@vtx.foundation`	Private name, valid without registration
Noise prologue	`vtx1`; in draft d03 `vtx1-d03`	Defined here
Media type (text form)	`application/vnd.vtx.jsonl`	Vendor tree, at freeze: `+jsonl` is not a registered suffix, and the standards tree needs a standards body
Core catalog	`tag:vtx.foundation,2026:vtx/catalog/core/1`	Tag URI, needs no registration
Well-known URI suffix	`vtx`, for browser clients (Section 2)	To register (RFC 8615)

Draft identifiers. Until the specification freezes, implementations use the draft-numbered identifiers, so two drafts never talk to each other by accident. This is draft d03. The labels inside hashes, such as `vtx1-approval` and `vtx1-oidc`, stay the same across drafts. Draft d02 moved the domain-scoped names, the SSH subsystem and the tag URIs of catalogs and schemas, from scailabs.ai to vtx.foundation.

Registries and their rules

Registry	Ranges	Rule
Frame types	0x00–0x0F common; 0x10–0x3F per channel, defined here or by the extension that defines the channel type; the extensions assign 0x22 WAIT on channel 0 (Viewers 1), 0x14 FETCH and 0x15 GONE on channel 5 (Transfer 1), 0x1F WINDOW on channel 2 (Windows 1) and 0x23 LAYOUT on channel 0 (Displays 1); 0x40–0x3FFF future versions; 0x4000 and up experimental	Experimental types MUST be gated by an extension capability
Well-known channels	0–5 assigned; 6–15 reserved	Assigned by specification only
Capability keys	Unqualified names for this spec; `name@domain` for extensions	Section 5 lists the core's keys; the extensions' keys are in `ext/registry.json`
Close codes	0x00–0x0D assigned by the core; 0x0E–0x11 assigned by extensions; 0x12–0x3F reserved; 0x40 and up implementation-defined	Section 14
Scene error codes	Strings listed in Section 7	New codes by specification

Registry	Ranges	Rule
Component catalogs	Tag or other URIs	Anyone may publish one; clients advertise what they render
Permissions	Strings listed in Section 12, with those the extensions add	New permissions by specification
Input device classes, gestures, accessibility actions	Lists in Section 11; `name@domain` for extension classes	New entries by specification; extensions need no registration
Auth methods	Strings listed in Section 5; the extensions add `rpk`, `device` and `enrol` (Devices 1) and `ticket` (Brokering 1)	New methods by specification; name@domain for extensions
AUTH_FAILED reasons	`credential`, `expired`, `revoked`, `unsupported`, `locked`; the extensions add `denied` (Devices 1) and `ticket` (Brokering 1)	New reasons by specification
Resource kinds	Strings listed in Section 10; the extensions add image, theme, catalog, audio and video kinds	New kinds by specification
Haptic effects	Strings listed in Section 11	New effects by specification; name@domain for extensions
Colour roles	`fg`, `bg`, `muted`, `accent`, `info`, `success`, `warning`, `critical`	By specification only, since every client must map them
Dynamic channel types	Strings listed in Section 4; the extensions add `audio` and `video` (Media 1), `transfer` (Transfer 1), `exec`, `forward` and `listen` (Exec 1), `device` (Devices 1), `viewers` (Viewers 1), `sensor` (Sensors 1), `broker` (Brokering 1), `drive` (Drives 1), `print` (Printers 1), `mediadev` (Media Devices 1), `smartcard` (Smart Cards 1), `usb` (USB 1) and `wasm` (WASM 1)	New types by specification; name@domain for extensions
Pointer shapes	CSS cursor keywords listed in `vtx-ops-1.schema.json`	New shapes by specification
Application interface kinds	Strings listed in Section 17, with those the extensions add	New kinds by specification

Every value the extensions assign in these registries is listed in `ext/registry.json` of the definitions bundle, which `tools/validate.py` checks against the CDDL on every run.

§17 Application interface

Applications do not speak the wire protocol. The server runs the application and talks to it in the text form: one JSON object per line, UTF-8 (JSON Lines). The server turns the application's ops into UPDATE frames and the client's input into lines for the application. A program that prints JSON, or an AI agent's tool runtime, can drive a VTX session this way. A server decides per application whether it uses this interface or runs the application under the VT engine (Section 8).

Transports.

- **stdio.** The server starts the application with standard input and output as the stream; standard error goes to the server's log.
- **WebSocket.** Subprotocol `vtx-app1` (`vtx-app1-d03` in draft d03), one line per text message, for applications that run elsewhere, such as an agent service. Which side connects, and how it authenticates (mutual TLS or a bearer token, for example), is the deployment's choice.

Lines from the application (`vtx-app-1.schema.json`). A line with `op` is a scene op (Section 7). Ops between `begin` and `commit` form one update; any other op is an update of its own. A `begin` without `commit` is discarded after 30 seconds or when the stream closes. A line without `op` is a control line, named by its `kind` (some ops, such as `chart` and `input`, have a `kind` prop of their own):

Kind	Purpose
`approval`	Ask the user to approve a request in the trusted zone (Section 12); answered by `zone_result`
`permission`	Ask for a permission (Section 12); answered by `zone_result`
`claim`	Set the name, icon and colour the trusted zone shows for the application (CLAIM)
`cancel`	Withdraw a pending approval, permission or clipboard read by `id`
`clipboard_write`	Write to the user's clipboard (Section 13)
`clipboard_read`	Ask to read the clipboard; answered by `clipboard_data`
`haptic`	Play a haptic effect on an input device (Section 11)
`resource`	Provide a glyph or font as base64 with its `format` and hash; the server checks the hash, then offers and serves it (Section 10)
`end`	End the session, with an optional `exit` status

Lines to the application (`vtx-app-events-1.schema.json`), all with `kind`:

Kind	Carries
`session`	`attached`, `detached`, `resumed`, `display`, `caps`, `locale` or `ended`, with the user, sign-in method, certificate (for `x509` and `tls-cert`), locale, time zone, Unicode version, accepted capabilities, display and input devices as they apply
`intent`	An INTENT, after the server has joined its parts and dropped duplicates (Section 6)
`input`	Any other input event: `event` names it (`key`, `pointer`, `resize` and so on) and `data` holds the frame's payload
`scene_error`	An op that failed: the error code with the op's state and index, or `parse` with the line number
`zone_result`	The answer to an approval or permission: `decision`, the request digest and, for `uv`, the WebAuthn assertion or the card signature
`clipboard_data`	The answer to `clipboard_read`, or `denied`

Rules.

- The server validates every line. An invalid op is not sent and produces `scene_error`; so does a line that is not JSON, with `code: "parse"` and its line number.

- The server numbers states, splits large updates, and handles credit, acknowledgement, retention and resume. The application sees one continuous session.
- The server checks approval digests and assertions before `zone_result` reaches the application, and passes on the assertion so the application can keep it with the request and its origin (Section 12).
- An application that closes its output ends the session as `end` would, without an exit status.

§18 Fallback rendering

A server must show layer 2 content to clients that cannot render it: a cell client without the core catalog, and a plain terminal reached over SSH or a serial line with no VTX client at all. The server then renders the content itself.

- For a cell client, the server renders each region into a grid of the client's width, using the fallback column of the core catalog table (Section 9), and each node's `fallback` for other types.
- For a plain terminal, it writes the same lines as VT bytes, using only what the terminal named by `TERM` supports.
- Widths follow the grapheme and width rules of Section 8. Text wraps at word boundaries; table columns shrink to fit and their cells end in … when cut.
- When the scene changes, the server redraws only the lines that changed, in place.
- On a cell client, the server keeps the grid's last row for prompts and answers. The rows above it show the content and scroll with PageUp and PageDown.

Input. Keyboard answers become the same INTENTs a component client would send, with the same `source`, `region`, `values` and `seq`, so the application cannot tell the difference. Menus take numbers, forms prompt for one field after another, checkboxes take y or n, and Escape cancels a form or closes an overlay.

Approvals on plain terminals. With no trusted zone, the server keeps the bottom line as a status line, with a scroll region (DECSTBM) above it, and asks there. A high-risk approval needs the request ID typed in full, not just y. `uv` approvals are answered `unsupported`. Such an approval is only as trustworthy as the server and the path to the terminal; applications that need proof of the user's intent require a VTX client.

§19 Conformance

Three conformance classes set the minimum each implementation supports. Everything above it is negotiated.

Class	Must support
Cell client	TLS over TCP; channels 0, 1, 2 and 4; grids and their ops; `input.keys`; the `token` and `tls-cert` auth methods; the trusted zone for approvals and permissions
Component client	All of the cell client, plus the core catalog, the clipboard channel and `webauthn`
Server	TLS over TCP and WebSocket; the `webauthn` and `token` auth methods; fallback rendering (Section 18); the application interface over stdio, unless the server embeds its application. The VT engine is optional

Profiles and extensions. The classes are the floor; the profiles name what a client renders above it. Profile 0 is a plain VT terminal reached through a gateway and needs no VTX client. Profile 1 is a cell client, Profile 2 a component client that also renders vector surfaces, images and themes, and Profile 3 a client that also renders pixel and window surfaces, audio and video, and offers redirection and WebAssembly components. No extension is mandatory: a peer offers each with its capability keys, and peers that share none still talk. `ext/registry.json` gives each extension the lowest profile whose clients can offer it, and its status: draft (not yet implemented), candidate (implemented, still open to change) or stable (frozen).

Extension	Lowest profile	Status	What it adds
Surfaces 1	2	draft	Vector, image and grid surfaces
Themes 1	2	draft	Branded themes on the root and regions
Composites 1	2	draft	Custom components from declarative catalogs
Media 1	3	draft	Audio and video streams, pixel surfaces, keyboard lock
Transfer 1	1	draft	Downloads, uploads, folders and files on the clipboard
Exec 1	1	draft	Remote commands and port forwarding
Devices 1	1	draft	Push wake-ups, device keys and enrolment
Viewers 1	1	draft	Shared sessions, roles, recording and calls
Sensors 1	2	draft	Device sensors, gamepad datagrams, surface accessibility
Windows 1	3	draft	The device's own windows for remote and VTX applications
Displays 1	3	draft	Multi-monitor sessions
Brokering 1	1	draft	App lists, launching and session routing
Drives 1	3	draft	The device's folders in a session
Printers 1	3	draft	Printing on the device's printers
Media Devices 1	3	draft	The device's microphones, speakers and cameras in a session
Smart Cards 1	3	draft	The device's smart card readers in a session
USB 1	3	draft	USB devices the user picks, in a session
WASM 1	3	draft	WebAssembly components, each drawing its own surface

Two bindings, a host binding and a gateway binding, sit beside the extensions and add nothing to the wire: Wayland Host 1 fixes how a Wayland compositor serves a session, and RDP Bridge 1 how a gateway puts Windows session hosts behind VTX.

Test vectors. `examples/vectors.json` in the definitions bundle holds known answers: QUIC varints, a CREDIT frame, an APPROVAL with its digest and WebAuthn challenge, an OIDC nonce, an update sent as two UPDATE frames, and an `x509` sign-in and card-signed approval made with a published test key. `tools/validate.py` recomputes or verifies each one; an implementation conforms on these points when it produces the same bytes.

Appendix: machine-readable definitions

The definitions ship as one MIT-licensed bundle, `vtx-v1-definitions`. Where the bundle and this text disagree, this text wins and the bundle is a bug.

File	Defines
`vtx-ops-1.schema.json`	One line of the text form: every scene op, with `open` expanded per node type for `region`, `grid` and the core catalog, and prefixed extension types. JSON Schema 2020-12 with only local references and closed objects, so constrained-decoding engines can enforce it on agents
`vtx-intent-1.schema.json`	One intent, client to application
`vtx-app-1.schema.json`	Any line an application writes: a scene op or a control line (Section 17)
`vtx-app-events-1.schema.json`	Any line the server writes to an application (Section 17)
`vtx-wire-1.cddl`	The CBOR payload of every frame type of the core, one rule per frame, prefixed by channel (`ctl-`, `scn-`, `in-`, `res-`, `zone-`, `clip-`, `any-`)
`ext/`	The eighteen extensions, each in its own folder with its CDDL, schemas and examples; the beta file with the `drafts` capability; the combined CDDL and schemas with the core and every extension; and `registry.json`, every identifier the extensions assign, with each extension's profile and status
`examples/valid/`	A fleet dashboard, a deploy form, a VT grid, templates, agent features (Markdown, extension types, focus, attention, styles), intents, and both directions of an application session
`examples/invalid/`	Lines that must be rejected: schema violations, scene-rule violations, malformed intents, and bad application lines in both directions
`examples/resources/`	Two glyph resources (a 1-bit bitmap and a vector path) that the examples refer to by their real SHA-256
`examples/vectors.json`	Known-answer test vectors (Section 19)
`tools/gen_schemas.py`	The single source of all four JSON Schemas
`tools/make_resources.py`, `tools/make_vectors.py`	Build the glyph resources and the test vectors
`tools/validate.py`	Runs every check below

Extension points. The CDDL marks where extensions may add map members or values as standard CDDL sockets (RFC 8610, Section 3.9), each named `…-ext`: `$$limit-ext`, `$$in-pointer-ext`, `$$in-gesture-ext`, `$$in-intent-ext`, `$$in-a11y-ext`, `$a11y-action-ext`, `$res-kind-ext`, `$permission-ext`; for sign-in `$auth-method-ext`, `$$auth-offer-ext`, `$$auth-ext` and `$auth-failed-ext`; and `$$hello-ext`, `$$ready-ext` and `$$approval-ext`. The core defines none of them. The eighteen extensions live under `ext/` in the bundle, beside `ext/vtx-1-all.cddl` and three combined schemas that hold the core and all of them at once; Section 19 lists them.

What the checks prove. `tools/validate.py` passes 2,543 checks with no failures: 444 for the core, listed below; 1,819 for the eighteen extensions (94 for Surfaces 1, 44 for Themes 1, 71 for Composites 1, 249 for Media 1, 240 for Transfer 1, 93 for Exec 1, 191 for Devices 1, 141 for Viewers 1, 114 for Sensors 1, 73 for

Windows 1, 82 for Displays 1, 55 for Brokering 1, 47 for Drives 1, 52 for Printers 1, 46 for Media Devices 1, 61 for Smart Cards 1, 57 for USB 1, 109 for WASM 1); 141 for the extensions together, their combined files and their registry; 67 for the Wayland Host 1 binding; and 72 for the RDP Bridge 1 binding. The core's checks prove that:

- every valid example line passes its schema, and every valid op stream, including the ops in an application session, also obeys the scene rules of Section 7;
- every invalid line is rejected, for the intended reason;
- the CDDL accepts a sample of every frame payload, the UPDATE frames built from the example streams and every example intent, and rejects 38 malformed samples, among them out-of-range gamepad axes, unknown gestures, incomplete device descriptors, a READY without `input_seq` and an `x509` signature without certificates;
- each glyph resource matches its hash and its declared size;
- QUIC varints round-trip, the CREDIT frame encodes to exactly the bytes in Section 3, and every test vector in `examples/vectors.json` recomputes from its inputs.

Review. A first independent review compared this text with the definitions and raised 25 issues, among them OIDC token relay, a flow-control deadlock, updates too large for one frame, and a Unicode version the client could not learn. A gap review then found 25 more (below). All 50 are resolved in draft d01 and in the bundle.

Resolved gaps (review of 11 October 2026)

A gap review of the previous draft and the outline found 25 open items: class A gaps would let two conforming v1 implementations disagree (18 items), class B left a v1 goal incomplete (5), and class C was missing from every release (2). All 25 are resolved in draft d01; the table keeps the record of what each one was. Four of them needed a decision, taken as follows: the reference web client runs on an independent origin and servers check `Origin`; identifiers carry the draft number until freeze; the roadmap gains push wake-ups, exec, port forwarding, session recording, raw keys and enrolment in v2, the A2UI component subset in v1, and brokering, app lists, multi-monitor and surface accessibility in v3; and v1 serial runs PPP under the TLS-over-TCP binding, with bare-metal ARQ experimental. The version labels in this record are those of the time; since draft d03, the v2 and v3 work is extensions of VTX 1.

#	Gap	Class	Fix in draft d01
1	No application interface (§7, §9, §12)	A	One JSON Lines stream between application and server (stdio, or WebSocket `vtx-app1`) carrying ops, zone requests and results, intents, input, scene errors and session events, with its own schema
2	Fallback rendering unspecified (§9)	B	Normative rendering of layer 2 into grids or VT bytes; typed answers become intents; on plain terminals, approvals live in a reserved status line
3	DANE has no procedure (§2, §5)	A	TLSA at `_port._udp` or `_tcp` of the SVCB target, DANE-EE and DANE-TA per RFC 7671, validated DNSSEC; a secure mismatch is fatal, otherwise WebPKI
4	URI grammar and SVCB details (§5)	A	RFC 3986 grammar without userinfo; `target` is path plus query; SNI uses the origin host; race `vtxq1` and `vtx1` per RFC 8305
5	Browser client trust (§2, §12)	B	Zone guarantees hold only for client code the server does not serve; serve the reference web client from an independent origin; servers check `Origin`

#	Gap	Class	Fix in draft d01
6	OIDC redirect and nonce (§5)	A	Fixed loopback and web redirect URIs; a host-bound nonce derived from a WELCOME seed, which browsers and load balancers can compute
7	Intents lost or doubled on resume (§6, §11)	A	Sequence numbers on INTENT, PASTE and composition commits; READY reports the last one delivered; the client re-sends the rest
8	Beta drafts indistinguishable on the wire (§1, §16)	A	Draft-numbered tokens such as `vtx1-d01` until freeze; bare tokens reserved for the frozen version
9	Grid size not tied to layout (§8)	A	Grid `size` (`fill` or `fixed`) and a RESIZE event per grid; the VT engine resizes the PTY on RESIZE
10	VT engine profile too thin (§8)	B	TERM, COLORTERM, LANG; mouse, focus and synchronized-output modes; colour queries; titles, bell, OSC 52; Kitty and sixel images deferred to v2
11	Grid terminal semantics (§8)	A	A per-style op and inline styles, a soft-wrap flag, scrollback clear, Hyper and AltGr modifiers, the unshifted key, palette theming rule
12	Focus rules (§9, §11)	A	Initial focus order, modal focus trap, and a server `focus` op that respects a user's recent focus change
13	Session end and roaming (§6, §14)	A	A `session_ended` code with exit status; default `linger` 0; reconnect only to the same host; resume from another device by the signed-in owner
14	Handshake state machine (§5)	A	Allowed frames per phase; 10 s HELLO and 120 s sign-in timeouts; at most 5 attempts; a fresh WebAuthn challenge per retry
15	Size limits and request floods (§7, §11–§13)	A	`max_update` and `max_clip` limits; continuation frames for PASTE and INTENT; at most 8 pending zone requests
16	Conformance and registries (§1, §16)	A	Conformance classes with mandatory auth methods; known-answer test vectors; the missing IANA entries and registries
17	Formatter and field semantics (§9)	A	Placeholder syntax, CLDR plurals, I-Regexp, lengths in graphemes, value types per field, RFC 3339 dates, and a time zone in HELLO
18	Catalog scoping and versions (§9)	A	A default catalog, prefixed types for other catalogs, immutable catalog URIs with `extends`, a core icon list
19	State after CAPS and resume (§4, §5, §10)	A	Channels 3 and 5 open when their capability is added; limits, locale and time zone renegotiable; resources re-offered after READY
20	Control characters and text direction (§8, §9)	A	Control characters invalid in scene text; bidi and zero-width characters shown in code; `lang` and `dir` props
21	Reaching a detached user (roadmap)	C	v2 `notify`: content-free push wake-ups (Web Push, APNs, FCM) when an approval is pending
22	Component states (§9)	B	`disabled`, `busy`, `readonly` and `error` props; a block `markdown` component; stable table row keys
23	Accessibility semantics (§7, §9)	B	Live regions for log and status, `description`, charts exposed as tables, grids as terminal documents, and role mappings per platform (Section 9)
24	Transport binding details (§2)	A	v1 serial means PPP carrying TLS over TCP; Noise shows the key fingerprint and disables `webauthn` and `oidc`; QUIC stream minimums and reset rules
25	SSH and RDS replacement features (roadmap)	C	v1: `vtx-a2ui` for the component subset. v2: remote exec, port forwarding, session recording, raw public keys and enrolment, A2UI media. v3: brokering, app lists, multi-monitor, surface accessibility