VTX WASM 1: WebAssembly components

Nov 15, 2026 · VTX 1 · extension · draft d01 · MIT licence · vtx.foundation/spec/v1/wasm-1/

§1 Status and scope

WASM 1 (an extension of VTX 1, draft d01, conformance profile 3) lets a server send small WebAssembly components that run on the user's device, each drawing into its own surface of the scene. It is for what the declarative layers cannot draw well or fast enough: a chart of thousands of points that follows the pointer, a signature pad, a waveform, a map, a board game. The server keeps the logic. The component keeps the drawing and tells the server what the user did.

It is the tier the outline set aside: "a sandboxed module that draws into its own surface and receives input for it", opt-in, Profile 3 only, with no network, no file system and no access outside its surface, within budgets the client sets, and always with a layer 2 fallback. It takes the capability key `wasm`, which the core keeps for it.

What this draft fixes:

- **Components, not bare modules.** A module is a WebAssembly component of one world, `vtx:wasm/surface@1.0.0`. The client links its three interfaces (`types`, `canvas`, `host`) and nothing else, so a component has no WASI, no sockets and no files to reach for. One that imports anything more is refused before it runs.
- **A fallback for every component.** A component surface carries layer 2 content as its children. The client shows it until the component runs, whenever the component stops, and whenever the user prefers it.
- **Named by content.** Modules travel by SHA-256, are cached per server, and can be signed by a publisher the client's policy trusts.
- **The client's budgets.** Module size, memory per instance, the number of instances, the time of one call and the size of a message are the client's to set. A component that overruns one is stopped and its fallback comes back.

Why not HTML and JavaScript: VTX is not a browser. A component gets no DOM, no network stack and no storage, and everything it can call fits on one page (Section 5). Components are language-neutral, since Rust, C, Go and Zig all compile to them, and every platform has a runtime for them: Wasmtime on desktops, an interpreter where a platform forbids generated code, and the browser's own engine for `vtx-web`.

Out of scope. Drawing on the GPU, sound, vector drawing lists, threads, components that talk to each other, and replaying components in a recording. Each is a later draft's work if a real application needs it.

§2 Capability and channel

The client offers `wasm` in HELLO only where the user or the organization's policy has turned components on for that server; it is off until then. The server grants it with `{}` in CAPS and opens one channel of type `wasm` (an even ID, as the server opens it), at most one per connection.

```
cap-wasm = {
  ? "worlds": [1*8 tstr],       ; client: the worlds it links (default ["vtx:wasm/
surface@1.0.0"])
  ? "max_module": 1..16777216,  ; client: largest module, bytes (default 4,194,304)
  ? "max_memory": uint,         ; client: linear memory of one instance, bytes (default
67,108,864)
  ? "max_instances": 1..64,     ; client: wasm surfaces open at once (default 8)
  ? "call_ms": 1..1000,         ; client: longest call into a component, ms (default
50)
  ? "max_message": 1..65536,    ; client: largest MSG either way, bytes (default 4,096)
  ? "signed": true,             ; client: only modules signed by a publisher its policy
trusts
}
```

Field	Default	What it bounds
`worlds`	`vtx:wasm/surface@1.0.0`	The worlds the client links. This draft has one.
`max_module`	4 MiB	One module's bytes, checked against MODULE before a byte travels
`max_memory`	64 MiB	One instance's linear memory: what the component declares, and all it may grow to
`max_instances`	8	Wasm surfaces open at once, across all modules
`call_ms`	50 ms	One call into a component: `init`, an event, a message or a frame
`max_message`	4 KiB	One MSG, either way
`signed`	off	Modules load only with a valid signature of a publisher the client's policy trusts

No permission prompt. A component reaches nothing the user would be asked about: it sees its own surface and talks only to the server that sent it. Instead, the trusted zone lists the components running in the session, by title and publisher, and stops any of them with one click (Section 9).

Closing the channel, from either side, stops every component, and every wasm surface shows its fallback. The server needs no STOPPED to know.

§3 Modules

A module travels once per server and then comes from the client's cache. MODULE names it and gives its title, world, size and SHA-256, and can carry a publisher's signature. The client answers in one of three ways: LOADED with `cached` when it already holds those bytes from this server, NEED when it wants them, or REFUSED when it will not run the module at all.

```
wasm-name = tstr .regexp "[A-Za-z0-9_.:-]{1,64}"     ; chosen by the server, never
reused in a connection
wasm-publisher = {
  "key": bstr .size 32,         ; the publisher's Ed25519 public key
  "sig": bstr .size 64,         ; Ed25519 over "VTX WASM 1 module", a zero byte and
sha256
}
wasm-module = {                                       ; 0x10 MODULE, S->C: a module the
scene will use
  "module": wasm-name,
  "title": tstr .size (1..80),  ; what the trusted zone calls it, e.g. "Throughput
chart"
  "world": tstr,                ; e.g. "vtx:wasm/surface@1.0.0"
  "size": 1..16777216,          ; bytes
  "sha256": bstr .size 32,
  ? "publisher": wasm-publisher,
}
wasm-need = { "module": wasm-name }                   ; 0x11 NEED, C->S: send me its
bytes
wasm-chunk = {                                        ; 0x12 CHUNK, S->C: after NEED,
in order
  "module": wasm-name,
  "offset": uint,
  "data": bstr .size (1..65536),
}
wasm-loaded = {                                       ; 0x13 LOADED, C->S: checked and
compiled
  "module": wasm-name,
  ? "cached": true,             ; from the client's cache for this server, without NEED
}
wasm-refused = {                                      ; 0x14 REFUSED, C->S
  "module": wasm-name,
  "reason": "world" / "size" / "signature" / "hash" / "format" / "imports" / "memory" /
"compile",
}
wasm-drop = { "module": wasm-name }                   ; 0x15 DROP, S->C: no surface
names it any more
```

Sending. After NEED, the server sends CHUNKs in order from offset 0, at most 64 KiB each, adding up to `size`. Every frame of every channel keeps flowing between them, so a large module never holds up the session.

Checking. The client refuses at the first check a module fails, in this order:

Check	Known at	Reason
Its world is one the client links	MODULE	`world`
`size` is at most `max_module`	MODULE	`size`
With `signed`: a publisher the client's policy trusts, and a valid signature	MODULE	`signature`
The bytes match `sha256`	the last CHUNK	`hash`
The bytes are a component, not a core module	the last CHUNK	`format`
Every import is an interface of its world	the last CHUNK	`imports`

Check	Known at	Reason
It exports every function of its world	the last CHUNK	`world`
The memory its core modules declare fits `max_memory`	the last CHUNK	`memory`
The client's runtime compiles it	the last CHUNK	`compile`

The first three need no bytes, so a client that fails one refuses straight after MODULE and no byte travels.

The cache. The client keeps modules by SHA-256, separately for each server identity. A cache shared between servers would let one server learn which modules the device had loaded for another, just by watching for NEED. LOADED with `cached` is only for bytes the client fetched from this server before, and the client checks them again against this connection's limits.

Signatures. A publisher signs the text `VTX WASM 1 module`, a zero byte and the module's SHA-256 with Ed25519. The signature says who built the module, not which server sends it. An organization requires one with `signed` and keeps the list of publishers it trusts in the client's policy, on the device; the list never reaches the server. The trusted zone names the publisher of every running module that has one.

Dropping. DROP frees a module that no surface names any more; the client may keep its bytes in the cache. A name is never used twice in a connection, so a new version of a module comes under a new name.

§4 At a glance

> **Diagram: a component's life · the module's checks above, each surface below.** Drawn in the PDF version, `18-wasm-1.pdf`.

The top row happens once per module and connection; the bottom row once per surface that names the module. Every way off the path to STARTED ends with the surface's fallback on screen.

§5 The component's world

The client links one world, `vtx:wasm/surface@1.0.0`, defined in `ext/wasm-1/vtx-wasm-1.wit`. It has three interfaces to import and seven functions to export, and nothing more is linked: a component that imports anything else is refused before it runs (Section 3).

```
/// VTX WASM 1 extension, draft d01: the world a WebAssembly component runs in
/// on a VTX client. A component draws into its own surface and gets the input
/// for it. It reaches nothing else: no network, no files, no clipboard, no other
/// node, no other instance. The client links these interfaces and nothing more.
/// Licence: MIT.
package vtx:wasm@1.0.0;

/// Shared types.
interface types {
  /// The surface's size in device pixels, and the display's scale
  /// (device pixels per px).
  record size {
    width: u32,
    height: u32,
    scale: f32,
  }

  enum pointer-kind { down, move, up, cancel, leave, wheel }

  /// A pointer event, in device pixels from the surface's top left corner.
  record pointer-event {
    kind: pointer-kind,
    x: f32,
    y: f32,
    /// The button that changed, as in the buttons of POINTER (v1):
    /// 1 primary, 2 secondary, 4 middle; 0 when none did.
    button: u8,
    /// Wheel only: the distance scrolled, in device pixels.
    dx: f32,
    dy: f32,
    pointer-id: u32,
    /// Milliseconds since the instance started.
    time: u64,
  }

  /// A key event. The code is a KEY code of v1; text is what the key typed.
  record key-event {
    code: string,
    text: option<string>,
    down: bool,
    repeat: bool,
    /// As in KEY (v1): 1 Shift, 2 Alt, 4 Control, 8 Meta, 128 AltGraph.
    mods: u8,
    time: u64,
  }
}

/// The surface's pixels.
interface canvas {
  /// Show a frame: width by height pixels, RGBA with 8 bits each, sRGB,
  /// premultiplied alpha, rows from the top. A frame whose size is not the
  /// surface's last size is dropped.
  present: func(width: u32, height: u32, pixels: list<u8>);
  /// Ask for one call of frame before the display's next refresh.
```

```
  request-frame: func();
  /// The pointer shape over the surface: a pointer shape name of v1.
  set-cursor: func(shape: string);
}

/// The way to the server, the clock and the screen reader.
interface host {
  /// Send bytes to the server: MSG on the wasm channel.
  post: func(data: list<u8>);
  /// Milliseconds since the instance started, in whole milliseconds.
  now: func() -> u64;
  /// What a screen reader says for the surface, and its current value.
  describe: func(label: string, value: option<string>);
}

/// A component that draws into one surface of the scene.
world surface {
  use types.{size, pointer-event, key-event};
  import canvas;
  import host;

  /// Start, with the config bytes from the scene and the surface's size.
  export init: func(config: list<u8>, size: size);
  /// The surface's size or scale changed; the next frame has the new size.
  export resize: func(size: size);
  export pointer: func(event: pointer-event);
  /// True when the component used the key; the client gets the others,
  /// so Tab moves the focus on.
  export key: func(event: key-event) -> bool;
  export focus: func(focused: bool);
  /// Bytes from the server: MSG on the wasm channel.
  export message: func(data: list<u8>);
  /// The call request-frame asked for, with milliseconds since the start.
  export frame: func(time: u64);
}
```

What the client gives. `canvas` shows the component's frames, asks for the next frame call and sets the pointer shape. `host` sends bytes to the server, tells the time since the instance started, and tells a screen reader what the surface shows. `types` holds the records both sides share.

What the component gives. `init` starts it with the surface's `config` bytes and size. `resize`, `pointer`, `key` and `focus` pass on what happens to the surface, `message` passes on the server's MSG, and `frame` is the call `request-frame` asked for. `key` answers whether the component used the key; the client acts on the others, so Tab always moves the focus on.

What it does not get. No WASI: no files, no sockets, no environment, no standard output, no time of day and no random source; a component that needs randomness seeds its own generator. No clipboard, no other node of the scene, no other instance, no threads and no shared memory. The clock counts whole milliseconds since the instance started: enough for animation, too coarse for timing attacks.

Calls. The client calls one function of an instance at a time and never calls in again while a call runs. Events arrive in the order they happened. Each call must return within `call_ms` (Section 7). Frame calls come only after `request-frame`, at most once per display refresh, and not while the surface is hidden.

Pixels. A frame is RGBA with 8 bits a channel, sRGB, premultiplied alpha, rows from the top, at the surface's size in device pixels: its size in px times the display's scale. A frame of any other size is dropped, since it was drawn before a resize. The client composes the frame into the scene where the surface is laid out; nothing a component draws reaches outside its surface.

Toolchains. Anything that builds components for a WIT world works, from Rust, C, Go or Zig. The example module in the bundle is a bar chart of about 3 KB, written in the WebAssembly text format and made into a component with `wasm-tools` (Section 10).

§6 In the scene

A component draws a surface of kind `wasm`. The surface names its module and can give the component starting bytes in `config`. Its children are its layer 2 fallback: the client shows them until the component runs, whenever it stops, and whenever the user prefers them.

```
{"op":"open","id":"tp","type":"surface","kind":"wasm","parent":"dash","module":"chart",
 "alt":"Throughput, last 24 hours","size":
[480,160],"config":"PTApJCIlOGGPrbi+srrCx8zSuJmAa1dH","mouse":"click","keys":"none"}
{"op":"open","id":"tp-table","type":"table","parent":"tp", ...}
```

Field	Meaning
`module`	The module whose component draws the surface, announced with MODULE and not dropped
`config`	Bytes for `init`, base64, at most 64 KiB
`alt`	What a screen reader says for the surface until the component calls `describe`
`size`	The surface's size in px, as for every surface. The component draws it in device pixels.
`mouse`	Which pointer events reach the component, as for every surface: `none`, `click`, `drag` or `move`
`pointer`	`lock` for pointer lock, under v1's rules
`keys`	`raw` (default): keys go to the component while it has the focus; `none`: no keys. Never system shortcuts.

`fit` and `zoom` do not apply: a component draws at the size it is laid out at, and zooming is its own business.

The fallback. A wasm surface has at least one child once the UPDATE that opens it is applied. While the component runs the children stay in the scene, hidden, and the server keeps them current, so the fallback that comes back after a stop shows the same data.

Starting and stopping. Once a surface's module is decided, the client answers for the surface with STARTED or STOPPED, and again whenever a patch sets its `module` or `config`, which starts a fresh instance:

```
wasm-started = {                                        ; 0x16 STARTED, C->S: the
component runs; its fallback is hidden
  "node": node-id,
  "module": wasm-name,
}
wasm-stopped = {                                        ; 0x17 STOPPED, C->S: the
fallback shows
  "node": node-id,
  "reason": "refused" / "trap" / "memory" / "time" / "user" / "fallback",
  ? "detail": tstr .size (1..200),   ; for the developer's log, e.g. the trap's message
}
wasm-msg = {                                            ; 0x18 MSG, both ways: to or from
a running component
  "node": node-id,
  "data": bstr .size (0..65536),
}
```

STARTED needs a module the client loaded and the user has not stopped. Removing the surface ends its instance without a word on either side. At most `max_instances` wasm surfaces are open at once; a server that wants more shows the rest as plain layer 2.

Messages. MSG carries bytes between the server and one running component, at most `max_message` of them, in either direction. What the bytes mean is the application's business: in the example, the server sends the chart's 24 bar heights and the chart sends back the hour the user clicked.

Input. While a component runs, pointer and key input over its surface goes to the component, and the client sends no POINTER, KEY or INTENT for it: what the user does there reaches the server only as the component's MSG. Two things never reach a component: the trusted zone's shortcut, and a key the component did not use, so Tab always leaves the surface.

Accessibility. The surface is one element of the client's accessibility tree, named by `alt` until the component calls `describe`. With a screen reader, the fallback is often the better view; the user can switch any wasm surface to its fallback, and the client then sends STOPPED with `fallback`. Switching back sends STARTED again.

Viewers. In a session with Viewers 1, each viewer's client runs its own instance on its own wasm channel. The server's MSG goes to each connection on its own, and a component's MSG names the viewer whose device sent it.

§7 Budgets and failure

A component that breaks a budget is stopped, never slowed down or warned: its instance is thrown away and the surface's fallback comes back. Every stop reaches the server as STOPPED with its reason, so the application can decide what to do next.

Reason	What happened	What follows
`refused`	Its module was refused (Section 3)	The fallback stays. The server can drop the module and send another.

Reason	What happened	What follows
`trap`	The component trapped: `unreachable`, an access out of bounds, a failed lowering	The fallback shows. The server can start a fresh instance by patching `module` or `config`.
`memory`	It tried to grow its memory past `max_memory`	As for `trap`
`time`	A call ran longer than `call_ms`	As for `trap`
`user`	The user stopped the module in the trusted zone	Every surface of that module stops, and the client starts it again for nothing in this connection
`fallback`	The user prefers the fallback for this surface	The client sends STARTED again if the user switches back

Time. The client interrupts a call that runs past `call_ms`, with the runtime's fuel or epoch interruption, and stops the instance. A component that has a long job splits it over frame calls. A surface that is hidden, scrolled away or in a minimized window gets no frame calls and can be suspended, but it is not stopped.

Memory. `max_memory` bounds one instance's linear memory, both what its core modules declare (a component that declares more is refused with `memory`) and what it grows to while it runs. Each instance has its own memory, even when several surfaces share a module.

Restarts. The server restarts a stopped component by patching the surface's `module` or `config`. A client may answer a component that keeps stopping for the same reason with that STOPPED at once, without running it again, so a broken module cannot drain a phone's battery in a loop.

What the user sees. The fallback, and nothing alarming: a stop is the application's problem, not the user's. The trusted zone keeps a line for each stop the user did not cause, such as "Throughput chart stopped: it took too long", for whoever looks.

§8 The application's side

An application hands the server a component once, opens wasm surfaces like any other node, and talks to each running component with two lines. The server does the rest: the hash, MODULE on every connection, NEED and the CHUNKs, and keeping track of which components run.

Line	Direction	What it does
`module`	to the server	Sends the clients a component: its name, title, world, bytes and, where it has one, the publisher's signature. Without `base64` it drops the module.
`wasm_message`	to the server	Bytes for one running component: MSG on the wasm channel
`wasm` event	to the application	A module `loaded` or `refused`, a component `started` or `stopped`, or a `message` it posted, with the viewer where Viewers 1 is in use

```
{"kind":"module","module":"chart","title":"Throughput chart","world":"vtx:wasm/
surface@1.0.0",
 "base64":"AGFzbQ0AAQAH2gEBQgkBcgMF...","publisher":{"key":"...","sig":"..."}}
{"kind":"wasm","action":"loaded","module":"chart"}
{"kind":"wasm","action":"started","node":"tp","module":"chart"}
{"kind":"wasm","action":"message","node":"tp","base64":"EQ=="}
{"kind":"wasm_message","node":"tp","base64":"PTApJCIlOGGPrbi+srrCx8zSuJmAa1dM"}
{"kind":"wasm","action":"stopped","node":"tp","reason":"time"}
```

Signing. The publisher signs a module where it is built, so the server and the application never hold the publisher's private key; the application passes the signature on with the bytes.

Keeping the fallback current. The application patches the fallback's nodes as the data changes, whether or not the component runs. A stop then shows the user the same numbers the component was drawing.

Lost messages. The server drops a `wasm_message` for a component that is not running, since the application learns of every start and stop from the `wasm` event.

§9 Security

A component is the server's code on the user's device. WASM 1 keeps it inside its surface and makes it no more dangerous than the server already is: what a component sees, the server would have seen as input anyway.

- **Nothing linked but the world.** No WASI, no sockets, no files, no clipboard, no environment. The client checks the imports before the first instruction runs and refuses a component that wants more.
- **Its own surface and nothing else.** A component's pixels land only inside its surface's box. It reads no other node and no other instance's memory, and the trusted zone sits outside the session's area, where no component can draw.
- **Input only where it belongs.** Pointer input over its surface, and keys while it has the focus. Never system shortcuts, keyboard lock or the trusted zone's shortcut, and every key it does not use goes back to the client.
- **The client's budgets.** Module size, memory, time per call, instances and message size are the client's to set, and breaking one stops the component (Section 7).
- **No timers for side channels.** The clock counts whole milliseconds since the start, and there are no threads and no shared memory to build a finer one from.
- **A cache per server.** One server cannot learn from NEED which modules another has sent.
- **Publishers.** Modules can be signed where they are built. An organization can require a trusted publisher, and the trusted zone names the publisher of each running module.
- **The user's stop.** The trusted zone lists the running components by title and publisher and stops any of them; a stopped module does not start again in that connection.

What the server learns. Only what the component posts. A component sees the pointer and keys over its surface, which the server would have received as POINTER and KEY without it, so posting them reveals nothing new.

What remains. The runtime is the boundary: a flaw in the client's WebAssembly runtime is a way out of the sandbox, so clients keep their runtime current and run components in a separate process where the platform allows. A component can draw anything inside its surface, a fake sign-in form among them. That is the risk of every pixel surface, and the answer is the same: credentials go only through the trusted zone. A component can also spend its whole budget on every call; the client runs instances at low priority and calls `frame` less often when the device is busy or on battery.

§10 Definitions and checks

The definitions are in `ext/wasm-1/`:

File	What it holds
`vtx-wasm-1.cddl`	The `wasm` capability and the nine frames of the wasm channel
`vtx-wasm-1.wit`	The world `vtx:wasm/surface@1.0.0`
`vtx-ops-wasm-1.schema.json`	Every op of the core, the first nine extensions, Windows 1 and Displays 1, plus the wasm surface
`vtx-app-wasm-1.schema.json`	Application lines: the above, every earlier control line, `module` and `wasm_message`
`vtx-app-events-wasm-1.schema.json`	Server lines to an application: every earlier one, plus the `wasm` event
`examples/components/`	The bar chart: `chart.wat` and the component `chart.wasm` (3,157 bytes), its core module `chart-core.wasm`, and the broken `chart-net` that imports a network interface
`examples/valid/`	The dashboard and cached traces, and the application's lines
`examples/invalid/`	32 broken traces, 6 broken wasm surfaces and 7 broken lines

`tools/gen_ext_wasm.py` writes the schemas and `tools/make_wasm.py` writes the examples. With `--assets`, `make_wasm.py` rebuilds the three `.wasm` files from their text with `wasm-tools`, through jco 1.11.1; the builds are byte for byte the same each time. While this draft was written, the chart also ran in jco's JavaScript host: it drew its bars, posted the index of the bar clicked, and drew again at a new size.

The traces. On QUIC, Marcel opens his network dashboard. The signed chart travels in two CHUNKs, runs over its table, takes the newest hour and reports the hour he clicks, 17:00. A site map that imports `vtx:wasm/net` is refused and its text stays. The chart runs out of time once and is restarted with fresh data; then Marcel stops it in the trusted zone, his click lands on the table, and the server drops the map. On WebSocket the next day, the chart comes from the cache and runs twice, for traffic in and out; Marcel switches one to its table and back, and closes the panel.

The validator reads the component binaries itself, section by section: whether they are components or core modules, their top-level imports and exports, and the memory their core modules declare. It then holds the client to the answer the bytes call for.

Part	Checks	What they prove
Schemas and CDDL	7	The schemas are valid; the CDDL parses alone and with every other extension and defines no rule twice; earlier schemas reject the wasm surface and the `wasm` event
The world	6	One world, importing `types`, `canvas` and `host` and using three of its types, exporting seven functions; the CDDL names it
The components	10	`chart.wasm` imports exactly the world's interfaces and types, exports its seven functions and declares one page of memory, so a client loads it; `chart-net.wasm` is refused for `imports`, `chart-core.wasm` for `format`, and the chart for `memory` and `world` under other limits
CDDL samples and ops	29	11 valid samples pass, 12 malformed ones fail, and 6 broken wasm surfaces fail the ops schema
Traces	35	Both traces keep every rule, the dashboard sends `chart.wasm` byte for byte, and each of 32 broken traces fails for its labelled reason, at the frame that breaks it
The application's side	22	15 lines are valid, with the fields each action needs, a module a client loads and a signature that verifies; 7 broken lines fail

The 32 broken traces cover the channel's opener, grant and count; a name used twice; NEED where the world, size or signature calls for a refusal (five cases); CHUNKs without NEED, out of order or past the size; LOADED for bytes that fail the hash, are a core module, import the network or need too much memory; LOADED from an empty cache and a refusal for the wrong reason; surfaces naming no module, without a fallback, over `max_instances` or without the capability; wrong STARTED and STOPPED, a surface left without either, MSG to or from a component that is not running or too long, pointer input reaching the server, a start after the user's stop, and DROP of a module in use.

The validator's section on the extensions as a whole gained three checks for the new registry entry. The whole bundle now runs 2,543 checks, all passing.

§11 Wire summary

Everything travels on one channel of type `wasm`, which the server opens:

Type	Frame	Direction	What it does
0x10	MODULE	S→C	Announces a module: name, title, world, size, SHA-256, and a publisher's signature
0x11	NEED	C→S	Asks for a module's bytes
0x12	CHUNK	S→C	Up to 64 KiB of them, in order
0x13	LOADED	C→S	Checked and compiled, after the CHUNKs or from the cache
0x14	REFUSED	C→S	Will not run it, with the reason
0x15	DROP	S→C	No surface names the module any more
0x16	STARTED	C→S	A wasm surface's component runs; its fallback is hidden
0x17	STOPPED	C→S	It does not run, or no longer; its fallback shows
0x18	MSG	Both	Bytes between the server and one running component

Besides the channel, WASM 1 adds the `wasm` capability, the surface kind `wasm` in the ops schema, the application lines `module` and `wasm_message`, and the `wasm` event. It adds no permission, close code or frame on a fixed channel.