HUBTALK

Hubtalk Web API Manual

How your system launches calling campaigns on its own and gets the results: how it works and the technical reference.

The current version is always on the site: docs.hubtalk.ai/v4/web-api · 17 sections.

Contents

How it works

Web API

The access key and getting ready

How a campaign runs

Consent, the registry and who is responsible

When something goes wrong

Technical reference

Calling the API

Creating a campaign

Example: retries by outcome

The campaign object, blockers and warnings

Campaign fields

Managing a campaign

The upload report

Individual calls

Calls from your website: Web SDK

Call outcomes: `call_status` and `status_reason`

Webhooks

Error codes and an end-to-end example

HOW IT WORKS

Web API

How your system launches campaigns on its own: where to start and where everything is documented.

This section explains what the Web API does, how to get an access key, how to launch a calling campaign through it, and how to read the responses.

The whole section in one file — "Hubtalk Web API". The file is updated together with the manual: handy to pass to a developer or load into an AI assistant.

⬇ Download MD ⬇ Download PDF

This section has two parts, written for different people:

- **How it works** requires no programming knowledge: what happens, who is responsible for what, and what has to be decided before the first request.
- **Technical reference** is for a developer: request paths, fields, response formats, error codes, and examples.

> **How to read the contract.** Everything named in a table is stable unless the table says otherwise. Two things are explicitly **not** part of the contract: `error.message` (English diagnostic prose) and the `message` field that may appear inside `blockers` / `warnings` items on `2xx` campaign responses. New fields and new codes are added over time — treat unknown fields as opaque, and an unknown code by falling back to `error.type + error.status`.

What this is and why

Normally a person creates a calling campaign: they log in to the cabinet, create a campaign, upload a file with phone numbers, and click **Launch**. That works well while campaigns are occasional.

The Web API is for when a campaign has to launch **by itself** — from your system, at the moment an event happens there. A contract goes past due, a ticket closes, a reminder date arrives — your system sends the platform a single request and the campaign starts. No person is involved at that moment.

> **The key property: under the hood this is the same product.** A campaign created by a request is no different from one created in the cabinet. The same pre-launch checks, the same number parsing, the same do-not-call registry, the same scheduler, the same per-number frequency caps, the same call recordings and reports. It appears in the cabinet in the common list and can be paused by hand. This is not "a second product for developers" — it is a second door into the same one.

Two working models — pick one

	Individual calls	**Campaigns**
What you send	one number at a time	a list of numbers (and more batches later, if you need them)
Request	`POST /v1/calls/phone`	`POST /v1/campaigns`
Who controls the pace	you	the platform
Who retries no-answers	you	the platform, **if you allowed retries**
Who watches working hours	you	the platform
Who checks the do-not-call registry	the platform	the platform
When it is convenient	your system already knows how to be a dialer	you want to hand over a list and get results

There is no need to mix them in one task: if you hand over a list as a campaign, the platform decides itself whom to call and when, and interfering "from above" with individual calls breaks its pacing calculation.

The rest of this page is mostly about **campaigns** — the reason the Web API is usually connected in the first place; individual calls are covered in Individual calls.

How it works

- The access key and getting ready — Issuing a key, what must be ready before the first request, and the pre-launch checklist.
- How a campaign runs — What happens after the request, how to read the response, and why the idempotency key is mandatory.
- Consent and responsibility — Consent attestation, the do-not-call registry, and what the platform does itself.
- When something goes wrong — The common refusals, what they mean and what to do about them.

Technical reference

- Calling the API — Base address, authentication, formats and the map of every request.
- Creating a campaign — `POST /v1/campaigns`: headers, request body, the order of refusals.
- The campaign object — What comes back, and how to read launch blockers and warnings.
- Campaign fields — Retries, numbers, dial binding, consent, registry screening, idempotency.
- Managing a campaign — Launch, pause, cancel, progress, statistics and adding batches of numbers.
- The upload report — What came back for each row of the file, and the row-level issue codes.
- Individual calls — `POST /v1/calls/phone`, the call object, reading, ending and listing.
- Call outcomes — The outcome axis: `call_status`, `status_reason` and how they fold into a verdict.
- Webhooks — The `call_ended` and `call_analyzed` events, signing, retries and delivery order.
- Error codes and an example — The full list of codes and a working scenario in one piece.

HOW IT WORKS

The access key and getting ready

Issuing a key, what must be ready before the first request, and the pre-launch checklist.

The access key: how to get it and what to do with it

The key is a long string your system uses to identify itself to the platform. It replaces a login and password: a program has no person to type those in by hand.

How to issue one

1. Log in to the cabinet with the account of an **administrator or developer** of your organization.
2. Open the **API keys** section in the left menu — it is a top-level menu item, not a subsection of **Settings**.
3. Click **Create key**.
4. Fill in:
   - **Name** — for you, so that you can later tell whose key it is ("Billing", "CRM production"). It is not used in requests.
   - **Requests per minute limit** — how many requests per minute this key is allowed. Empty or `0` means the platform-wide limit (usually 60 per minute; the exact value in force is shown next to the field). The limit protects you too: a broken integration will not flood the queue.
5. Click **Create**.

> **The secret is shown exactly once.** The dialog displays a string of the form `cfk_…`. Copy it and hand it to your developer over a secure channel — the platform stores only a hash and cannot show the secret again. If the secret is lost, revoke the key and issue a new one.

What else you can do

- **Change the request limit** — with the pencil icon in the list. The secret does not change, so the integration needs no reconfiguration; the new limit applies from the next request.
- **Revoke a key** — the row stays in the list marked as revoked (needed for audit), but requests with it stop working immediately.

What the key cannot do

- **Reach beyond your organization.** The key sees only its own campaigns, agents, lines, and uploads. Anything outside that boundary is not "forbidden" for it — it simply does not exist for it: the answer is the same as for an object that never existed.
- **Replace telephony configuration.** The key does not create lines and does not bind agents to them: an administrator does that in the cabinet.

Why it matters WHO issued the key

The platform records which employee issued the key and when. This is not an audit formality but a legal basis: launching a campaign requires a signature under the attestation that you have the subscribers' consent, and the signature is placed by the key — that is, by a program. A program's action holds only as long as it can be tied to the person who took on the obligation. That person is the one who issued the key. The "who and when" record is stored in the signature itself and is not assembled after the fact.

Practical conclusion: **issuing a key is not a technical detail.** It should be issued by someone entitled to answer for calling campaigns on behalf of the organization.

What must be ready before the first request

Everything listed here is configured in the cabinet once; without it a campaign will not start, and the platform states plainly what is missing.

What	Who sets it up	What you get from it
SIP trunk with an outbound subgroup	an administrator in Telephony	`trunk_id` and the subgroup's `route_id`. The caller-ID pool, the allowed destinations and the call limits belong to the **subgroup**, not to the trunk
Agent is published	your flow developer	`agent_id` (the agent name); it is the published version that calls, a draft is never dialed
Agent is bound to an outbound subgroup	an administrator	otherwise there is nothing to call from: the platform does not pick a line for you
Market package (for the US)	platform support	without it, geography and the legal calling window cannot be checked
The organization's calling window	a platform administrator	outside this window the platform does not dial, whatever hours a campaign names
Key is issued	an administrator or developer, the API keys section	the `cfk_…` secret, see The access key
Webhook endpoint registered	an administrator, on the agent: editor → **Settings** tab → **Webhooks (post-call)**	your URL and the signing secret shared with you out of band

A note on binding: if the agent is bound to exactly one outbound subgroup, **you do not have to specify anything** — the platform finds it by the agent. If there are several, it will not choose for you and will ask you to name the line explicitly: guessing here would mean calls from someone else's number and against someone else's quota, and you would find out from the invoice.

Checklist before the first launch

1. The agent is created and **published**; a test call from the cabinet works.
2. The agent is bound to an outbound subgroup; if there are several, you know which one to name.
3. For the US: the market package is installed and the list of numbers contains the geography columns.
4. The organization's calling window is set the way you need it; a campaign's working hours cannot widen it.
5. The key was issued **by someone entitled to answer for calling campaigns**, and the secret was handed to the developer over a secure channel and stored in your secret storage.
6. The key's request limit was set deliberately.
7. Your system generates a **new** idempotency key for every new campaign and every batch.
8. The campaign purpose (`classification`) is set explicitly and matches what the agent actually talks about.
9. It is decided who retries no-answers — you or the platform — and the number of attempts is set accordingly (the default is one).
10. The first campaign was launched on a **short list of your own numbers**, the response was read in full, and the per-row report was reviewed.
11. You verified that the campaign is visible in the cabinet and can be paused by hand — this confirms the integration works with the same product.
12. It is clear who on your side reads progress and what they do with blockers if a campaign fails to start overnight and nobody notices.

What follows is for the developer: how to call the API, which request does what, and what comes back.

Contract changes from 14 October 2026 — check how you parse responses

- **The transcript holds only what the caller heard.** Agent replies that never played are no longer included in `transcript` — in the `call_ended` and `call_analyzed` webhooks, in `GET /v1/calls/{id}` and in the transcript columns of reports. The `pending` and `unspoken` marks no longer arrive. This also applies to old calls when a webhook is retried. See the call object.
- **Two new `status_reason` values**: `caller_abandoned` (`not_connected`) — an inbound caller hung up before the agent answered, formerly `no_speech`; `no_media` (`error`) — the callee answered but no audio came, unless the agent has this detection turned off (it is on by default). See call status.
- **The dialing speed is any whole number from 1 to 200**, not one of fixed tiers, and it can now be changed on an existing campaign with `PATCH /v1/campaigns/{id}`.

The version number did not change. A parser that does not recognize a `status_reason` value should treat it by its `call_status` bucket.

Three contract changes from 22 September 2026 — check how you parse responses

Along with the removal of the shared dialing ceiling (see dialing speed), three fields left the responses. None of them is replaced by anything else — they are simply gone, because so is the mechanism they reported on:

- `warnings[]` on campaign creation carries no `rate_clamped` code;
- the `GET /v1/campaigns/{id}/progress` response has no `rate.clamped` field;
- in the campaign limits, `max_dials_per_minute` is replaced by `calls_per_minute_tiers`, the list of available tiers.

A parser that reads `rate.clamped` without checking whether the key is there will break on this. `calls_per_minute` always equals what you sent (or set later with `PATCH`).

HOW IT WORKS

How a campaign runs

What happens after the request, how to read the response, and why the idempotency key is mandatory.

How a request-launched campaign runs

Step 1. Your system sends a single request. It contains the campaign name, the agent name, the dialing speed, the working hours, the purpose of the campaign, the consent attestation, and the list of numbers with the data for the conversation.

Step 2. The platform parses the list. The same parsing as for a file in the cabinet: number normalization, duplicate detection, required-column checks, do-not-call registry matching, geography checks for the US. None of this is skipped or simplified on the API path.

Step 3. The platform checks whether launching is allowed. Is the agent published, is there a line, is the window in the past, is the attestation signed, did all numbers end up in the do-not-call registry.

Step 4. If everything is fine, the campaign starts. With the same scheduler as campaigns from the cabinet: at your speed, within your working hours, retrying no-answers by your rules, and under the per-number frequency caps in force in your organization.

Step 5. You read the response. It tells you what happened: whether the campaign started, how many rows were accepted, and what is wrong with the rejected ones.

There is no separate "button press" in this scheme: one request does all three things at once — creates the campaign, accepts the numbers, and launches it.

How to read the response

A response always arrives, even when the campaign did not start. You read it from one field and two lists.

`status` **— the single answer to "did it start?":**

Value	Meaning
`running`	the campaign is dialing
`scheduled`	everything is fine, waiting for the window to open
`draft`	it did not start; the reason is in the `blockers` list

`blockers` **— why it did not start.** These are not "request errors" but the state of affairs: the agent was unpublished, numbers matched the do-not-call registry, the window is in the past, there is no signature under the consent attestation. Each reason has its own code — the same one the cabinet shows on the launch screen.

> One subtlety worth knowing in advance: if you **read the campaign later** with a separate request, `blockers` answers the question "what would a launch check report right now", not "why is the campaign not working". For a running campaign that list is never empty — at the very least it contains "a signature under the attestation is required", because the signature is given per launch request and is not stored as a standing state. The question "is it dialing" is answered only by `status`.

`warnings` **— non-blocking notices.** They block nothing and are present on practically every campaign: for example "no start was set, dialing begins right away" and "a frequency policy is in force" with the figures for how many times a day and a week one number may be called.

`import` **— what happened to the list of numbers.** How many rows were seen, accepted, accepted with warnings, rejected, how many duplicates, and for which reasons (grouped by reason code with the numbers of the first rows). `dnc_screen` states separately: how many readable rows were screened, how many were not, how many rows matched the

registry and how many distinct numbers that is; every match carries the row, the original phone, the name, the reason and the registry record.

> **The campaign and the report are kept in every outcome.** Even when the campaign did not start, it remains in the cabinet and the per-row report is available: this is the only way to understand what is wrong with the data, and deleting it along with the failure would destroy exactly what is needed to fix it.

One rule that removes the guesswork

An action that changed nothing is always an error. A successful response means the state in the response body *is* the new state. Therefore:

- a create request answers with success even if the campaign did not start: the campaign and the report really were created, and `status` honestly says `draft`;
- a separate "launch" request that launched nothing answers with an **error** — and names the reason inside the error, so you do not have to ask a second time;
- a repeated "pause" on an already paused campaign is **not** an error: the state is already what you asked for.

Retrying a request: why the idempotency key is mandatory

The network sometimes drops the response after the platform has already done everything. At that moment your system does not know whether the request arrived, and usually repeats it. For a single call that is annoying; for a list of ten thousand numbers it means **calling everyone twice**.

That is why campaign creation **and** adding numbers to a campaign both require the `Idempotency-Key` header — an arbitrary string your system generates for every new campaign or batch (for example, the task identifier in your system). From there the platform sorts it out itself:

- **same key, same body** — the same response as the first time is returned; no second campaign appears;
- **same key while the first request is still running** — `409 idempotency_key_in_flight`: wait for the response, do not retry again;
- **same key but a DIFFERENT body** — `422 idempotency_key_reuse`. This guards against the nastiest failure: without it you would receive the response to someone else's request and conclude that the platform ignored your settings;
- **the request was rejected before the campaign was created** — the key is released immediately, and a corrected request with the same key goes through normally. If the request was rejected *after* the campaign was created (the list could not be stored, say), the key stays taken and you find the campaign with `GET /v1/campaigns`.

Keys are remembered for 24 hours. Practical conclusion: **a new campaign means a new key, a new batch means a new key.** Retrying with the previous key makes sense only for literally the same request.

HOW IT WORKS

Consent, the registry and who is responsible

Consent attestation, the do-not-call registry, and what the platform does itself.

Consent attestation and the purpose of the campaign

Before launch the platform requires you to attest that you have the subscribers' consent to be called. In the cabinet this is a checkbox with legal text; through the API it is the `attestation` field in the request. Without it the campaign is still created, but stays a draft with the reason "a signature is required".

The signature is placed by the **key**, and the human link is the employee who issued the key (The access key). The signature is tied to a specific version of the data: an attestation given for one list of numbers does not carry over to the next one. Upload a new list or add a batch of numbers and you attest again.

The purpose of the campaign (`classification`) — what exactly you are calling about:

Value	What it covers	Consent tier
`marketing`	offers, advertising, surveys	`pewc` — prior express written consent (the strictest)
`servicing`	servicing: reminders, notifications, statuses	`pec` — prior express consent
`debt_collection`	debt collection	`pec` — prior express consent

The purpose determines the **consent tier** (`consent_tier`) — which legal text you actually sign. That is why the platform does not guess it: if you do not specify it, `marketing` is used, the strictest option. The response returns both the purpose and the resulting tier, so you can see which text you signed.

The do-not-call registry

The platform finds numbers from your organization's registry **before** the first call and lists them one by one. Every readable row of the list is screened before duplicates are removed: three rows carrying the same number are therefore three matched rows but one excluded number. Rows whose phone could not be read count as "not screened", and a clean result is always returned as an explicit `clean` status rather than left to be inferred from an empty list. The launch stops at that point.

What happens next is your decision:

- **keep the block** (`dnc: "block"` , the default) — the matched numbers are not dialed;
- **confirm the campaign** (`dnc: "skip"`) — the platform records this decision in the audit log exactly as it records the button in the cabinet, noting which upload **and which exact composition of matches** it applies to. A new registry record against the same list resets the confirmation, and the launch check shows the refreshed list of matches.

If **all** dialable numbers turned out to be in the registry, there is nothing to confirm — the launch stops unconditionally. Immediately before the launch the platform reads the live registry again and stores its snapshot together with the signature; after the launch the campaign card and the match report show exactly that snapshot, even if the registry changes later. Numbers that land in the registry while the campaign is already running are still not dialed — the campaign progress counts them separately.

> For batches of numbers added to an existing campaign (Consent and responsibility) the rule is harsher: a registry match does not "stop the launch" but **refuses the whole batch** — you cannot confirm calling such numbers through the add-calls path.

What the platform does itself and what stays with you

The platform does: number and variable parsing, duplicates, the do-not-call registry, geography and the legal calling window (US), the organization's calling window, dialing pace, no-answer retries, per-number frequency caps, contact time

zones, call recordings, result parsing.

You keep: deciding when to launch a campaign and with which list; storing the link between your own entities and the campaign; reading progress if you need a status in your interface; receiving call results (webhooks per call work the same as for individual calls).

About no-answer retries. By default a campaign makes **one attempt** per number and calls nobody back. If you want the platform to retry by itself, set the number of attempts and the pause between them (`retry_config`). If your own system manages the redials — it sends the number again as its own retry — leave one attempt: otherwise the redial becomes two-level, yours plus the platform's. A number sent again that the campaign has already dialed is accepted as a new call and dialed again.

About working hours. If you did not give the campaign working hours, that does not mean "dial round the clock": the platform dials within your organization's calling window (and in the US — within the legal window of the subscriber's jurisdiction). A campaign's hours may only **narrow** that window; an attempt to set wider hours is refused immediately.

Adding numbers to a campaign that already exists — possible, with a separate request (`POST /v1/campaigns/{id}/calls`): the batch carries the numbers and the signature, while the agent, the line, the working hours and the retries come from the campaign. It works for any campaign that is not closed — a draft, a scheduled, a running or a paused one. Rules worth knowing in advance:

- **a batch is accepted whole or refused whole.** One unreadable row and not a single number from the batch is dialed (`calls_rejected`); one number from the do-not-call registry and the batch is refused as well, but with its own code (`calls_dnc_matched`), so your system can tell "fix the data" from "remove these people from your base". The answer carries the parse of the refused batch with the upload number and the first rows; the full report is fetched as for any upload;
- **a batch is accepted even when the campaign is not dialing right now** — while it is paused, for instance. The answer says the numbers were accepted and names the reason for the standstill (`campaign_paused`); dialing continues after a resume;
- **an ordinary campaign closes itself when the queue drains** — and after that it accepts no numbers at all. For a campaign that is meant to wait for numbers, create it **open to call additions**: `accepts_call_additions: true` with a mandatory end date, no more than 30 days out. Such a campaign may be created empty, filled with batches, and closed either by cancelling it or by letting its end date arrive. The flag is set at creation and cannot be changed;
- **no more than 1000 numbers per batch** — split larger ones.

Deliberately not provided — do not look for these capabilities, they do not exist:

- **a separate "campaign finished" event.** State is read with a progress request or from the campaign card;
- **a separate campaign results endpoint with transcripts.** Results are read as ordinary calls;
- **different permissions for different keys.** A key either exists or it does not; restricting access per area is not provided.

HOW IT WORKS

When something goes wrong

The common refusals, what they mean and what to do about them.

Errors arrive as a machine code plus an English explanation. **Rely on the code:** the text of the explanation may change. These are the ones you meet most often in practice; the full list is in Error codes.

Code	What happened	What to do
`idempotency_key_required`	no idempotency key was sent	add the header, see How a campaign runs
`idempotency_key_reuse`	the same key with a different body	take a new key for the new campaign or batch
`idempotency_key_in_flight`	the first request with this key is still running	wait for its answer, do not retry
`campaign_agent_unknown`	there is no agent with this name in your organization	check the agent name; an unpublished agent is a different case — it produces a draft, not a rejection
`dialing_route_unresolved`	the agent is not bound to any line	bind it in Telephony
`dialing_route_ambiguous`	the agent is bound to several lines	name the line explicitly, see Dialing binding
`dialing_binding_incomplete`	a number or subgroup was named without a line	send `trunk_id` as well, or send none of the three
`campaign_timezone_unknown`	the time zone was not recognized	use a name like `Asia/Almaty`; offsets like `+05:00` are not accepted
`campaign_classification_invalid`	the campaign purpose is not from the list	use one of the three values in Consent and the campaign purpose
`campaign_window_invalid`	the end is earlier than the start	check the window
`campaign_spec_invalid`	settings refused: the speed is outside 1–200, or the retries, working hours, duplicate action or name length are invalid	check the settings against the reference
`working_hours_wider_than_law`	the hours are wider than the organization's calling window or the legal window	narrow the hours, or do not set them at all
`launch_blocked`	the launch failed the checks	the reasons are listed inside the error; a do-not-call blocker carries the refreshed list of matches
`campaign_transition_invalid`	the transition is impossible from the current state (resuming a cancelled campaign, say)	check the campaign status
`market_package_not_installed`	your organization has no market package, so the numbers cannot be screened	contact support
`dnc_screen_required`	an upload of the old format, with no per-row registry screening	upload the numbers again
`report_filter_invalid`	an invalid `only` value on the upload report	drop the filter or pass `only=dnc_match`

Code	What happened	What to do
`calls_rejected`	a batch of added numbers contains a row with a data error — the whole batch was refused	fix the rows from the parse in the error and send the batch again with the same idempotency key
`calls_dnc_matched`	a batch of added numbers contains a number from the do-not-call registry — the whole batch was refused	remove those numbers from your base (the list is in the upload report with `only=dnc_match`) and send it again
`campaign_closed`	the campaign is finished or cancelled and accepts no more numbers	create a new campaign
`calls_required` / `calls_batch_too_large`	the batch is empty or larger than 1000 rows	send between 1 and 1000 numbers per request
`attestation_invalid`	the batch attestation is not exactly `{"confirmed": true}`	send exactly that field, without `import_id`
`attestation_import_id_required`	a separate launch did not name the upload the signature covers	pass the `import_id` from the create response
`campaign_open_ends_at_required` / `campaign_open_ends_at_too_far`	a campaign open to call additions has no end date, or one beyond 30 days	name an end date within the horizon
`campaign_auto_speed_ends_at_required` / `campaign_auto_speed_working_hours_required`	auto speed needs an end date, and a campaign over several days also needs working hours	send `ends_at` and `working_hours`, or turn `auto_speed` off
`campaign_not_found` / `import_not_found`	there is no such campaign or upload in your organization	check the identifier

TECHNICAL REFERENCE

Calling the API

Base address, authentication, formats and the map of every request.

Calling the API

- **Base URL:** `https://{{HUBTALK_FQDN}}` — every path below starts with `/v1`. Substitute the host your own cabinet runs on for `{{HUBTALK_FQDN}}` — for example `app.hubtalk.ai`.
- **Format:** JSON in the request and in the response. Time values are **epoch seconds**.
- **Machine-readable schema:** `GET https://{{HUBTALK_FQDN}}/openapi.json`, with an interactive explorer at `https://{{HUBTALK_FQDN}}/docs`. The schema is authoritative for **requests**; success responses are published there as untyped objects, so the tables on this page are the response contract.

Authentication

Every `/v1` request carries the key:

```
Authorization: Bearer YOUR_API_KEY
```

The secret (`cfk_` + hex) is issued in the cabinet and shown once (The access key); the platform stores only a SHA-256 hash. A revoked key stops working immediately — the key record is re-read on every request. A key is scoped to the organization it was issued in: anything outside that boundary is indistinguishable from missing (`404`), never `403`.

Failure	Status	`error.type`	`error.code`
missing or malformed `Authorization` header	401	`authentication_error`	`api_key_required`
unknown key	401	`authentication_error`	`api_key_unknown`
revoked key	401	`authentication_error`	`api_key_revoked`
per-key rate limit exceeded	429	`rate_limit_error`	`api_key_rate_limit_exceeded` (`Retry-After: 60`)

Per-key rate limit. Each key has a sliding 60-second window over all its `/v1` requests. The limit is set per key at creation and can be **changed later** in the cabinet without rotating the secret (it applies from the next request). A key limit of `0` means the platform default — a deployment setting, 60 requests per minute unless the operator changed it; an operator may also disable the limit. This is one of **four distinct 429 sources**.

One error envelope

All `/v1` errors look the same:

```
{
  "error": {
    "type": "rate_limit_error",
    "code": "dial_queue_full",
    "status": 429,
    "message": "Dial queue is full; retry after the interval in Retry-After"
  }
}
```

HTTP status	`error.type`
400, 422	`invalid_request_error`
401	`authentication_error`

HTTP status	`error.type`
403	`permission_error`
404	`not_found_error`
409	`conflict_error`
429	`rate_limit_error`
503	`service_unavailable_error`
500, 502	`api_error`

`error.code` names the exact cause within the type and is **always** present: when no more specific cause is registered, `code` equals `type`. New codes are added over time — handle an unrecognized code by falling back to `type` + `status`.

Two additive fields appear only on specific campaign refusals and nowhere else: `error.blockers` on `409 launch_blocked` and `error.import` on `422 calls_rejected` / `422 calls_dnc_matched`. Every other error carries exactly the four fields above.

> Classify the error on `error.type` + `error.status`, and the exact cause on `error.code`. **Never match on `error.message`** — it is a human-readable English diagnostic, looked up by code at the API boundary, and not part of the contract. On `422` schema-validation errors (`code: "invalid_request"`) `message` is an **array** of field-level issues in FastAPI/pydantic form (`{"type", "loc", "msg", "input"}`) rather than a string.

Refusals carry their headers through the envelope: `Retry-After` on 429s is always delivered.

Rate limits and `429`

A `429` comes from four independent sources. All carry `error.type: "rate_limit_error"` — distinguish them by `error.code`:

`error.code`	Source	What to do	`Retry-After`
`api_key_rate_limit_exceeded`	sliding 60-second window on all requests made with the key	slow down your overall request rate	yes, always `60`
`dial_queue_full`	the platform's dial queue is at its depth ceiling; **the call was not created**	repeat the same request with the **same** `Idempotency-Key` after the indicated interval	yes (queue estimate, 1–60 s)
`subgroup_concurrency_limit`	the subgroup's concurrent-call limit is exhausted	wait for active calls to finish; no honest estimate exists, use your own backoff	no (deliberate)
`subgroup_rate_limit`	the subgroup's per-minute initiation limit is exhausted	wait out the minute window	yes (`60`)

Subgroup limits default to 10 concurrent calls and 30 initiations per minute unless the administrator configured other values; for a trunk used under a grant the effective limit is the smaller of the subgroup's and the grant's.

Request map

Campaigns

Method and path	Purpose
`POST /v1/campaigns`	create the campaign, hand over the numbers, and launch — one request
`GET /v1/campaigns`	cursor-paginated list of campaigns (short views)
`GET /v1/campaigns/{id}`	current state of one campaign

Method and path	Purpose
`GET /v1/campaigns/{id}/launch-check`	`{blockers, warnings, dnc_screen}` before acting
`POST /v1/campaigns/{id}/launch`	launch a campaign that stayed a draft
`POST /v1/campaigns/{id}/calls`	add a batch of numbers to an existing campaign
`POST /v1/campaigns/{id}/pause`	stop dialing new numbers
`POST /v1/campaigns/{id}/resume`	continue (launch checks run again)
`POST /v1/campaigns/{id}/cancel`	cancel irreversibly
`PATCH /v1/campaigns/{id}`	change the dialing speed or turn on auto speed
`GET /v1/campaigns/{id}/progress`	contact states, holds, pace, do-not-call and frequency counters
`GET /v1/campaigns/{id}/rate-history`	the auto-speed recomputation log

Uploads

Method and path	Purpose
`GET /v1/imports/{import_id}`	the upload summary plus the live or frozen screening
`GET /v1/imports/{import_id}/report?format=csv`	per-row report, `csv` or `xlsx`
`GET /v1/imports/{import_id}/report?only=dnc_match`	only the do-not-call matches, `csv` or `xlsx`

Lines

Method and path	Purpose
`GET /v1/dialing-routes?agent_id=…`	which trunks and subgroups are available to the agent

Individual calls

Method and path	Purpose
`POST /v1/calls/phone`	one outbound call
`POST /v1/calls/web`	web call: a session for the browser, see Calls from your website
`GET /v1/calls`	list of calls with filters and a cursor
`GET /v1/calls/{call_id}`	the full call object
`GET /v1/calls/{call_id}/turns`	per-turn latency metrics of a finished call
`POST /v1/calls/{call_id}/end`	end a live call

TECHNICAL REFERENCE

Creating a campaign

`POST /v1/campaigns`: headers, request body, the order of refusals and the successful response.

One request does three things: it creates the campaign, accepts the planned calls and — if every launch check passes — starts dialing. There is no separate "start" step in the happy path.

Headers

Header	Required	Notes
`Authorization: Bearer YOUR_API_KEY`	yes	
`Content-Type: application/json`	yes	
`Idempotency-Key: <unique string>`	**yes**	Unlike `POST /v1/calls/*`, it is not optional here: a retry after a network timeout would create a second campaign and dial the whole list twice. Missing or blank → `422 idempotency_key_required`. The protocol is in Idempotency of campaign requests.

Request body

Unknown top-level keys are ignored.

Field	Type	Required	Default	Description
`name`	string	yes	—	Campaign name shown in the cabinet; 1–80 characters after trimming (else `422 campaign_spec_invalid`).
`agent_id`	string	yes	—	Agent **name**, resolved inside your organization. Unknown name → `422 campaign_agent_unknown`. An existing but unpublished agent is accepted: the campaign is created as a draft with the `agent_not_published` blocker.
`calls_per_minute`	integer	yes, unless `auto_speed: true`	—	Any whole number from 1 to 200 (outside that range → `422 campaign_spec_invalid`; a value that is not a whole number → `422 invalid_request`). Can be changed later with `PATCH /v1/campaigns/{id}`. The value is applied as sent: there is no deployment ceiling that would cut it. Optional with `auto_speed: true`: if sent, it is kept as the manual speed the campaign returns to when auto speed is turned off.
`auto_speed`	boolean	no	`false`	Auto speed: the platform picks the speed to get through the base before `ends_at`, recomputed every 10 minutes, never above the subgroup limit, the organization's concurrent call limit and 200 per minute. Requires `ends_at` (`422 campaign_auto_speed_ends_at_required`); if the campaign window crosses midnight in its time zone, also `working_hours` (`422 campaign_auto_speed_working_hours_required`). A real boolean only: `1` or `"true"` → `422 invalid_request`.

Field	Type	Required	Default	Description
`planned_calls`	array	no	`[]`	The numbers to dial, see below. Empty means the campaign is created with nothing to dial (blocker `no_pending_contacts`, unless `accepts_call_additions` is `true`).
`starts_at` / `ends_at`	number | null	no	`null`	Campaign window. A `null` start means "now". `ends_at` must be later than `starts_at` (otherwise `422 campaign_window_invalid`). A start **date** in the past (by calendar date in the campaign zone) blocks the launch (`starts_date_past`); a past `ends_at` blocks it too (`ends_at_past`).
`timezone`	string	no	`""`	IANA zone name (`Asia/Almaty`) in which the window and the working hours are interpreted. Omitted → the organization's zone. Unknown zone → `422 campaign_timezone_unknown`; fixed offsets are not accepted.
`working_hours`	array	no	`[]`	Schedule rows `{"days": [0..6], "start": "HH:MM", "end": "HH:MM"}` (`days`: 0 = Monday, non-empty; `start` strictly before `end`; a row never crosses midnight — a night shift is two rows). Invalid form → `422 campaign_spec_invalid`. A campaign may only **narrow** the floor that applies to your organization: under a US market package the legal calling window of the contacts' jurisdictions, otherwise the organization's calling window; a wider row → `422 working_hours_wider_than_law`. **Empty means the floor itself applies** — the campaign narrows nothing, and dialing is limited by the legal window (warning `working_hours_law_only`) or by the organization's calling window (warning `working_hours_org_window_only`). Dialing is never round-the-clock.
`call_strategy`	`"standard"` | `"expert"`	no	`"standard"`	`standard` retries on a fixed ladder; `expert` decides per outcome from `retry_config.rules`.
`retry_config`	object	no	`{"attempts": 1, "interval_s": 900}`	See Retry configuration. **The default is one attempt — no redials.**
`duplicate_action`	`"reject"` | `"keep_first"` | `"keep_last"`	no	`"reject"`	What to do when the same phone appears twice in `planned_calls` (and inside a `calls` batch).
`classification`	`"marketing"` | `"servicing"` | `"debt_collection"`	no	the organization's default, else `"marketing"`	What the campaign calls about. It decides the **consent tier** the attestation is signed under (`marketing` → `pewc`, `servicing` and `debt_collection` → `pec`). An omitted value takes the organization's default campaign classification; if the organization has none, `marketing`, the strictest tier. Anything else → `422 campaign_classification_invalid`.

Field	Type	Required	Default	Description
`use_contact_timezone`	boolean	no	`false`	Dial each contact in its own zone (from the contact's `timezone` column) instead of the campaign's.
`trunk_id`, `route_id`, `from_number`	string	no	resolved	Dialing binding, see below.
`attestation`	object	no	—	`{"confirmed": true}` — the consent attestation required before dialing. Optional at creation: without it the campaign is created as a draft with the `attestation_required` blocker.
`accepts_call_additions`	boolean	no	`false`	Keep the campaign open to added batches of numbers: a drained queue does not complete it; only `ends_at` or `cancel` does. Requires `ends_at` (`422 campaign_open_ends_at_required`) no further ahead than the operator's horizon, 30 days by default (`422 campaign_open_ends_at_too_far`). Set at creation only.
`dnc`	`"block"` | `"skip"`	no	`"block"`	What to do about numbers matching the do-not-call registry. Anything else → `422 dnc_decision_invalid`.

Fields that exist on the campaign but are **not** settable through `/v1` (`frequency_override`, cabinet-only presets) are silently ignored if sent.

Order of refusals on creation

Before the campaign exists (the idempotency slot is released, a corrected retry with the same key is fine):

1. `422 invalid_request` — body fails the schema (missing `name`, `agent_id` or `calls_per_minute` with auto speed off, wrong types).
2. `422 idempotency_key_required`, `422 dnc_decision_invalid`.
3. `422 idempotency_key_reuse` / `409 idempotency_key_in_flight` / replay of the earlier answer.
4. `422 campaign_window_invalid`, then `422 campaign_auto_speed_ends_at_required` / `campaign_auto_speed_working_hours_required`, then `422 campaign_open_ends_at_required` / `campaign_open_ends_at_too_far`.
5. `422 campaign_agent_unknown`, `422 campaign_timezone_unknown`.
6. Dialing binding: `422 dialing_route_unresolved` / `dialing_route_ambiguous` / `dialing_binding_incomplete`.
7. `422 working_hours_wider_than_law`.
8. Repository validation: `422 campaign_spec_invalid` (speed outside 1–200, retry configuration, working-hours form, duplicate action, name length), `422 campaign_classification_invalid`.

After the campaign exists (the slot is **kept** — a retry with the same key answers `409 idempotency_key_in_flight`; find the campaign with `GET /v1/campaigns`):

9. Planned calls: `409 market_package_not_installed`, `502 contact_file_storage_unavailable`, `422 contact_rows_rejected` (the rows could not be stored at all — distinct from rows rejected by the parser, which are reported inside `import` with a `201`).
10. Launch: `409 campaign_transition_invalid` (unexpected state race).

Success response — `201 Created`

The response is **always** `201` when the campaign was created, whether or not dialing began. The single source of truth about that is `status`.


```
{
  "campaign_id": "cam-b9b536a86e",
  "status": "running",
  "created_via": "api",
  "launched_via": "api",
  "name": "june collections",
  "agent_id": "collections-agent",
  "timezone": "Asia/Almaty",
  "timezone_source": "request",
  "starts_at": null,
  "ends_at": null,
  "accepts_call_additions": false,
  "calls_per_minute": 10,
  "call_strategy": "standard",
  "classification": "marketing",
  "consent_tier": "pewc",
  "retry_config": { "attempts": 1, "interval_s": 900 },
  "duplicate_action": "reject",
  "use_contact_timezone": false,
  "working_hours": [{ "days": [0, 1, 2, 3, 4], "start": "09:00", "end": "19:00" }],
  "binding": { "trunk_id": "trk-31d4dc745a", "route_id": "rt-out-a", "from_number": "77000000001" },
  "import": {
    "import_id": "imp-353d158dd2",
    "status": "accepted",
    "rows_seen": 2,
    "accepted": 2,
    "warned": 0,
    "rejected": 0,
    "duplicates_found": 0,
    "dnc_matched": 0,
    "refusal_code": "",
    "issues": {},
    "dnc_screen": { "status": "clean", "checked_rows": 2, "unscreened_rows": 0, "matched_rows": 0,
"suppressed_contacts": 0, "fingerprint": "20e2…", "items": [], "has_more": false, "hidden_rows": 0,
"all_dialable_suppressed": false, "as_of_launch": true, "screened_at": 1788941847.27, "audit_id": 2, "decision":
"clean", "import_id": "imp-353d158dd2" },
    "dnc_matches": []
  },
  "dnc_screen": { "status": "clean", "as_of_launch": true, "…": "same object as import.dnc_screen" },
  "blockers": [],
  "warnings": [
    { "code": "starts_immediately", "message": "…" },
    { "code": "frequency_policy_active", "rolling_7d_cap": 7, "daily_cap": 2, "source": "platform_default",
"message": "…" }
  ]
}
```

TECHNICAL REFERENCE

Example: retries by outcome

A complete `POST /v1/campaigns` request with expert logic: its own retry interval for busy, no answer, voicemail and short calls.

Each call outcome can have its own pause before the next dial: an expert logic rule with the `retry_in` action and the `params.interval_s` parameter. Below is a ready request and what each part of it does.

The whole request

The campaign is created and launched with one `POST /v1/campaigns` request.

```
BASE_URL="https://app.hubtalk.ai"   # the host your cabinet runs on

curl -X POST "$BASE_URL/v1/campaigns" \
  -H "Authorization: Bearer YOUR_API_KEY" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: campaign-2026-10-22-01" \
  -d '{
    "name": "Payment reminder — October",
    "agent_id": "payment_reminder",
    "calls_per_minute": 10,
    "timezone": "Asia/Almaty",
    "working_hours": [
      { "days": [0, 1, 2, 3, 4], "start": "09:00", "end": "19:00" }
    ],
    "classification": "debt_collection",
    "call_strategy": "expert",
    "retry_config": {
      "attempts": 6,
      "interval_s": 3600,
      "rules": [
        { "if": "busy",              "then": "retry_in",          "params": { "interval_s": 600,  "max_matches": 3
} },
        { "if": "no_answer",         "then": "retry_in",          "params": { "interval_s": 5400, "max_matches": 2
} },
        { "if": "voicemail_reached", "then": "retry_in",          "params": { "interval_s": 7200, "max_matches": 2
} },
        { "if": "short_call",        "then": "retry_in",          "params": { "max_duration_s": 15, "interval_s":
3600 } },
        { "if": "no_speech",         "then": "wait_for_pass_end", "params": { "max_matches": 1 } },
        { "if": "user_declined",     "then": "stop_calling" }
      ]
    },
    "attestation": { "confirmed": true },
    "planned_calls": [
      { "phone": "+77001234567", "name": "Aigerim", "variables": { "debt": "15000" } },
      { "phone": "+77007654321", "name": "Daniyar", "variables": { "debt": "8200" } }
    ]
  }'
```

The answer is `201` with `campaign_id`, `status`, `blockers` and `warnings`. `status: "running"` means dialing has started; `draft` means something blocks it, and the reason is in `blockers`. The full response is in Creating a campaign.

What each rule does

Each rule is a pair "if outcome → then action". The interval counts from the end of the call and, for that contact, replaces the campaign interval.

Outcome (`if`)	Action (`then`)	After	Times per contact
`busy` — line busy	`retry_in` — redial after a pause	10 min (`600`)	3
`no_answer` — not picked up	`retry_in`	1 h 30 min (`5400`)	2
`voicemail_reached` — voicemail	`retry_in`	2 h (`7200`)	2
`short_call` — conversation under 15 s (`max_duration_s`)	`retry_in`	1 h (`3600`)	no rule limit, only the campaign `attempts`
`no_speech` — connected, but the contact stayed silent	`wait_for_pass_end` — redial on the next pass	—	1
`user_declined` — the contact declined the call	`stop_calling` — do not call again	—	—

The `voicemail_reached` rule also fires on `voicemail_suppressed` — voicemail was detected, but leaving a message was not allowed. Every `if` value is a `status_reason` code from Call outcomes, plus `short_call` and `goal_not_achieved`. An outcome that no rule matches is decided by the general outcome table, as in `standard`: voicemail without a rule of its own is redialed after the campaign `interval_s` (Call outcomes). That is why the example has its own `voicemail_reached` rule; once its `max_matches` is used up, voicemail is redialed by the general table while dials are left. To stop calling voicemail, use `stop_calling`.

Rules and limits

- `attempts` **is the overall cap on dials per contact, the first one included**, 1 to 30. Rules do not extend it: when the dials run out, the contact is closed even if a rule has matches left.
- `interval_s` **of** `retry_in` is whole seconds from `60` to `86400` (1 minute to 24 hours). It is required with `retry_in` and forbidden with the other actions, otherwise `422 campaign_spec_invalid`.
- `max_matches` is how many times the rule may fire for one contact, 1 to 30, default 1. `short_call` **has none** — sending it there is refused with `422 campaign_spec_invalid`.
- `max_duration_s` **of** `short_call` is required, at least `10` seconds.
- **No retry on a permanent reason.** For `invalid_destination` and `spam_blocked` on every market, and for `user_declined` and `screener_declined` under the US market package, only `stop_calling` is allowed. On other markets `user_declined` is temporary and a retry on it is allowed — the example sets `stop_calling` on purpose.
- **One "condition + action" pair — once.** A duplicate is refused.
- **Expert mode without a single rule** cannot be launched: the `rules_empty` blocker.
- **The calling window, the do-not-call registry and the frequency caps** apply to every retry. A retry that does not fit before the end of the campaign (`ends_at`) is not scheduled.
- **The** `rule_interval_exceeds_end` **warning** means a rule interval is not shorter than the campaign window left: that retry will not happen. It does not block the launch.
- **Do not send a rule** `id` — the server assigns it (`r-xxxxxxxx`) and returns it in the response.

For dialing to start right away

- `Idempotency-Key` is required, a new one for every new campaign. Repeating the request with the same key after a timeout does not create a second campaign.
- `attestation: {"confirmed": true}` is the contacts' consent attestation. Without it the campaign is created as a draft with the `attestation_required` blocker. See Consent, the registry and who is responsible.
- **The agent is published.** Otherwise — a draft with the `agent_not_published` blocker. `agent_id` is the agent's **name**.
- **The agent has Detect answering machines on.** Without it there is no `voicemail_reached` outcome, and the voicemail rule never fires.
- **Do not send** `trunk_id`, `route_id` **and** `from_number` — the platform finds the line from the agent (Dialing binding).

- `working_hours` **only narrows** the organization's calling window; a wider row → `422 working_hours_wider_than_law`.

See also

What	Page
Headers, request fields, `call_strategy`, order of refusals, response example	Creating a campaign
`retry_config` and rules: `if`, `then`, `params`	Campaign fields → Retry configuration
Outcome codes for `if`	Call outcomes
Launch blockers and warnings	Campaign object
Expert logic in the cabinet	Campaign launch

TECHNICAL REFERENCE

The campaign object, blockers and warnings

What comes back about a campaign, and how to read launch blockers and warnings.

The campaign object

Returned by `POST /v1/campaigns`, `GET /v1/campaigns/{id}`, `POST …/launch`, `POST …/pause`, `POST …/resume`, `POST …/cancel`.

Field	Type	Description
`campaign_id`	string	Identifier (`cam-…`).
`status`	string	`draft` | `scheduled` | `running` | `paused` | `completed` | `cancelled`.
`created_via`, `launched_via`	string	`api` | `web` | `""`. `launched_via` is stamped on the launch transition and stays `""` for a draft.
`name`, `agent_id`	string	As stored (name trimmed).
`timezone`	string	The **effective** IANA zone.
`timezone_source`	string	`request` when you named a zone, `organization` when it was inherited. Identical on every path (create, read, list).
`starts_at`, `ends_at`	number | null	Epoch seconds.
`accepts_call_additions`	boolean	
`calls_per_minute`	integer	The rate the campaign dials at, calls per minute. Equals the value you sent or set later. On an auto-speed campaign — the manual speed it returns to when auto speed is turned off.
`auto_speed`	boolean	`true` — the platform picks the speed, see auto speed.
`auto_rate_current`	integer | null	The pace a running auto-speed campaign dials at after the last recomputation. `null` on a manual campaign and on a campaign that is not running.
`auto_rate_needed`	integer | null	How many calls per minute are needed to get through the base before `ends_at` — before any limit. Above `auto_rate_current` means the limits keep the campaign from finishing in time. `null` wherever `auto_rate_current` is.
`call_strategy`, `classification`	string	
`consent_tier`	string	`pewc` | `pec` — derived from `classification`, never stored; the tier whose wording your attestation is signed against.
`retry_config`	object	Normalized form; rule ids assigned.
`duplicate_action`, `use_contact_timezone`, `working_hours`		Normalized echoes of the settings.
`binding`	object	`{"trunk_id", "route_id", "from_number"}` — the binding in force; `route_id` is `""` when it is resolved from the trunk at launch rather than stored.
`import`	object | null	The upload summary of **this request's** `planned_calls` (creation only). `null` when `planned_calls` was empty and on every read path — the summary lives at `GET /v1/imports/{import_id}`.
`dnc_screen`	object | null	Screen of the campaign's current (launch) import: live while a draft, frozen after launch. `null` when the campaign has no import.
`blockers`, `warnings`	array	Present on create, read and `launch`; **absent** on `pause` / `resume` / `cancel`.

How to read the creation response

1. `status` — `running` (dialing), `scheduled` (window is in the future) or `draft` (did not start). Nothing else in the body means "it started".
2. A non-empty `blockers` on **creation** ⟹ `status: "draft"`. The campaign and its import report are **kept**. Fix what the blockers name and call `POST /v1/campaigns/{id}/launch`.
3. `import` is `null` when `planned_calls` was empty; `blockers` then carries `no_pending_contacts` (unless the campaign is open to call additions).
4. `warnings` are non-blocking. Two of them are present on practically every campaign: `starts_immediately` (no future `starts_at`) and `frequency_policy_active` (the per-number frequency caps in force) — the latter replaced by `frequency_limits_disabled` when those caps are switched off.
5. `timezone_source` is `request` when you named a zone and `organization` when it was inherited. It reads the same on every path — the create response, `GET /v1/campaigns/{id}` and the list all agree.

Blockers and warnings

`blockers` and `warnings` are computed by the same launch check the cabinet runs. Each item is an object with a machine-readable `code`; most blockers also carry `field` — the request field the finding is about.

Shape depends on the path — read only what is guaranteed:

Where	Item shape	Notes
`2xx` campaign object (`POST /v1/campaigns`, `GET /v1/campaigns/{id}`, `POST …/launch`, `GET …/launch-check`)	`{"code", "field"?, "message", …extra}`	`message` is a **Russian, human-oriented diagnostic and not part of the contract** — never match on it or show it to end users. Extra context fields per code are listed below and are stable.
`409 launch_blocked` (`error.blockers`)	`{"code", "field", "dnc_screen"?}`	Codes and fields only; do-not-call blockers additionally carry the refreshed `dnc_screen`.
`201` of `POST …/calls`	`{"code", "field"}`	Codes and fields only (`field` may be `""`).

> **Reading `blockers` on `GET /v1/campaigns/{id}` and `launch-check`.** On the read paths the list answers "what would a launch check report right now", computed **without** an attestation and regardless of the current status. Consequences: a running campaign always shows `attestation_required` (the signature is given per launch request, not stored as a standing state), and a running campaign whose queue has drained shows `no_pending_contacts`. So on read paths a non-empty `blockers` does **not** imply `status: "draft"` — use `status` for that, and read `blockers` as the checklist a future `launch` / `resume` would have to pass.

Blocker codes

Code	`field`	Extra fields	Meaning
`no_pending_contacts`	`contacts`		No contacts waiting to be dialed. Not raised for a campaign with `accepts_call_additions: true`.
`agent_not_published`	`agent_id`		The agent has no published version.
`binding_invalid`	`trunk_id`		Trunk, subgroup or caller ID of the binding is unavailable (agent not in the subgroup, empty caller pool, trunk grant missing…).
`starts_date_past`	`starts_at`		The start **date** (in the campaign zone) is in the past. A past *time* on today's date is only the `starts_immediately` warning.
`ends_before_starts`	`ends_at`		`ends_at` ≤ `starts_at`.

Code	field	Extra fields	Meaning
`ends_at_past`	`ends_at`		`ends_at` already passed.
`policy_unavailable`	`market_profile`		The dialing policy (market package) of your organization is not installed; contact support.
`contacts_missing_geo`	`contacts`		US market: the upload's contacts carry no geolocation.
`contacts_geo_rejected`	`contacts`	`rejected`, `filename`	US market: rows of the current upload rejected for unusable location. Lifted only by a new upload.
`dnc_screen_required`	`contacts`	`dnc_screen`	The current import has no recorded do-not-call source-row projection: upload the contacts again.
`dnc_matches`	`contacts`	`dnc_screen`, `registry_changed`	Matches with the registry and no valid `skip` decision for this exact screen. `registry_changed: true` means a skip existed but the registry changed since.
`all_contacts_suppressed`	`contacts`	`dnc_screen`	Every pending contact is on the registry; cannot be skipped.
`attestation_required`	`attestation`		No consent attestation in this request.
`attestation_stale`	`attestation`		`attestation.import_id` is not the campaign's current import.
`rules_empty`	`retry_config`		`expert` strategy with no rules.
`rule_permanent_retry`	`retry_config`	`rule_id`	A rule retries a permanent outcome (`invalid_destination`, `spam_blocked`; `user_declined` and `screener_declined` under the US profile).
`signature_columns_missing`	`contacts`	`agent`, `variables`	The contacts lack a column for one or more of the agent's signature variables (those marked **In signature** on the agent); `variables` lists them. Checked against the agent's current published version; on `POST …/calls`, against the batch's own columns. Variables outside the signature are not checked.
`campaign_paused`	`status`		`POST …/calls` **only**: the batch is accepted but nothing dials until `resume`.
`auto_speed_ends_at_required`	`ends_at`		Auto speed without `ends_at`.
`auto_speed_working_hours_required`	`working_hours`		Auto speed on a campaign over several days without working hours.

Warning codes

Code	Extra fields	Meaning
`starts_immediately`		No future `starts_at`: dialing begins at launch.
`frequency_policy_active`	`rolling_7d_cap`, `daily_cap`, `source` (`organization` | `platform_default`)	The per-number frequency caps in force for this campaign. Present whenever the limits are on — that is, on every campaign except the case below.

Code	Extra fields	Meaning
`frequency_limits_disabled`	`source` (`campaign` | `organization`)	The configurable frequency limits are switched off, and `source` says by which setting. Comes **instead of** `frequency_policy_active`, never alongside it. On the US market profile, Reg F and the state caps keep applying; the Kazakh profile has no legal layer.
`working_hours_law_only`	`field: "working_hours"`	US market, empty schedule: only the legal window applies.
`working_hours_org_window_only`	`field: "working_hours"`, `start`, `end`, `source`	Empty schedule: the organization's calling window `start`–`end` applies.
`retry_schedule_exceeds_end`	`lost`	The retry ladder is longer than what is left of the campaign window; `lost` retries will not happen.
`variable_case_mismatch`	`variable`, `column`	A column was matched to a signature variable ignoring case/separators.
`variable_partial`	`variable`	The signature variable is filled for some rows only.
`rule_action_unsupported`	`rule_id`	An `expert` rule uses an action the engine does not execute; the general outcome table decides. All three actions — `wait_for_pass_end`, `stop_calling` and `retry_in` — are executed, so this warning is not raised today.
`rule_goal_without_criteria`	`rule_id`	A `goal_not_achieved` rule while the agent defines no success criterion — the rule never fires.
`rule_budget_masked`	`rule_id`	A rule's `max_matches` is not below the campaign's attempts — unreachable.
`rule_interval_exceeds_end`	`rule_id`, `interval_s`, `window_s`	A `retry_in` rule interval is not shorter than the campaign window left — that retry will not happen.
`auto_speed_exceeds_subgroup_limit`	`field: "trunk_id"`, `needed`, `limit`	Auto speed: finishing before `ends_at` needs `needed` calls per minute, while the subgroup allows `limit` — part of the base will not be reached.
`auto_speed_exceeds_concurrency_limit`	`field: "ends_at"`, `needed`, `limit`, `slots`	The same, limited by the organization's concurrent call limit: `slots` calls carry about `limit` per minute.
`auto_speed_exceeds_platform_limit`	`field: "ends_at"`, `needed`, `limit`	The same, limited by the platform ceiling of 200 per minute.
`manual_speed_exceeds_subgroup_limit`	`field: "trunk_id"`, `speed`, `limit`	A manual speed above the subgroup limit: the dials above it will be refused and retried.
`manual_speed_exceeds_concurrency_limit`	`field: "limits"`, `speed`, `limit`, `slots`	A manual speed needs more concurrent calls than the organization has: `slots` carry about `limit` per minute.

TECHNICAL REFERENCE

Campaign fields

Retries, numbers, dial binding, consent attestation, registry screening and idempotency.

Retry configuration

`retry_config` is normalized on save and echoed back normalized.

Field	Type	Default	Rules
`attempts`	integer	`1`	Total dial attempts per contact, 1–30. `0` reads as `1`; values above 30 are clamped to 30; negative → `422 campaign_spec_invalid`. **With the default, no number is redialed.** The cabinet campaign wizard has the same limit — up to 29 retries (30 dials) — and offers interval presets from 1 minute to 7 days; the API accepts any positive `interval_s`.
`interval_s`	integer	`900`	Pause between attempts in seconds. Must be `> 0` when `attempts > 1`, else `422 campaign_spec_invalid`.
`retry_mode`	`"regular"` | `"custom"`	`regular`	`custom` uses the per-retry ladder `intervals_s`.
`intervals_s`	array of integers	—	Seconds before each retry in `custom` mode (fitted to `attempts − 1`).
`rules`	array	—	`expert` only. Each rule: `{"id"?, "if", "then", "params"?}`. `if` is a `status_reason` value (Call outcomes) or `short_call` / `goal_not_achieved`; `then` is `wait_for_pass_end`, `stop_calling` or `retry_in`; `params`: `max_matches` (1–30, default 1) for every condition except `short_call` — there it is refused with `422 campaign_spec_invalid`, and only the campaign's `attempts` limit it — `max_duration_s` (≥ 10, required) for `short_call`, `interval_s` for `retry_in` — whole seconds from 60 to 86,400 (1 minute to 24 hours), required with that action and forbidden with the others, otherwise `422 campaign_spec_invalid`. With `retry_in` the next attempt comes no earlier than `interval_s` after the call ends: for that contact the rule interval replaces the campaign interval, and the calling window, the do-not-call registry and the caps apply as to any retry. A `retry_in` rule saved earlier without `interval_s` takes the campaign interval. Ids are assigned by the server (`r-xxxxxxxx`); ids you send that are not echoes of existing ones are replaced. A bare array of rules is accepted as the legacy form.

A different retry interval per outcome

With `call_strategy: "expert"`, each outcome can have its own pause before the next dial: a `retry_in` rule with `params.interval_s`. An outcome that no rule matches is decided by the general outcome table, as in `standard` (Call outcomes): for example, voicemail without a rule of its own is redialed after the campaign `interval_s`.

```
{
  "call_strategy": "expert",
  "retry_config": {
    "attempts": 6,
    "interval_s": 3600,
    "rules": [
      { "if": "busy", "then": "retry_in", "params": { "interval_s": 600, "max_matches": 3 } },
      { "if": "no_answer", "then": "retry_in", "params": { "interval_s": 5400, "max_matches": 2 } },
      { "if": "voicemail_reached", "then": "retry_in", "params": { "interval_s": 7200, "max_matches": 2 } },
      { "if": "short_call", "then": "retry_in", "params": { "max_duration_s": 15, "interval_s": 3600 } },
      { "if": "no_speech", "then": "wait_for_pass_end", "params": { "max_matches": 1 } },
      { "if": "user_declined", "then": "stop_calling" }
    ]
  }
}
```

Rule	What happens
`busy`	Redial in 10 minutes, up to 3 times.
`no_answer`	Redial in 1 hour 30 minutes, up to 2 times.
`voicemail_reached`	Redial in 2 hours, up to 2 times.
`short_call`	The call lasted under 15 seconds — redial in 1 hour.
`no_speech`	The contact stayed silent — redial once, on the next pass of the list.
`user_declined`	Stop calling this number.

- `attempts` caps the dials per contact across all rules together: with `6`, no number is dialed more than six times, whatever the rule budgets add up to.
- A retry that would fall after the campaign's `ends_at` is not scheduled; the `rule_interval_exceeds_end` warning names such rules (Campaign object).
- A retry action on a permanent reason (`invalid_destination`, `spam_blocked`) is refused.
- `expert` with an empty `rules` array cannot be launched: it yields the `rules_empty` blocker.

The complete request with this configuration is in Example: retries by outcome.

The numbers: `planned_calls`

Each object is one row of a contact list, and **its keys are column names**. Every column the file upload understands is understood here: market-package columns (`contact_zip`, `contact_state`, `contact_city`), contract-group columns (`contract_id`, `role`), `timezone`, `name`, and any variable your agent's flow reads. A nested `variables` object is sugar and is flattened into the same columns (a top-level key wins over the same key inside `variables`). Values are converted to text (booleans → `true` / `false`, objects → compact JSON), so leading zeros survive. Bare `zip`, `state` and `city` are ordinary variables, not location.

```
{
  "planned_calls": [
    {"phone": "+77001234567", "name": "Aigerim", "variables": {"debt": "15000"}},
    {"phone": "+77007654321", "name": "Daniyar", "contact_zip": "73301"}
  ]
}
```

A `phone` column is mandatory in every row; a value with fewer than ten digits is rejected as `phone_unreadable` — the **row** is rejected, not the request. Row-level issue codes are in The upload report.

Dialing binding

You have no internal trunk or subgroup identifiers, so the platform resolves them: **omit `trunk_id`, `route_id` and `from_number` together**, and the subgroups bound to your agent are looked up across every trunk visible to your organization.

Outcome	Result
exactly one subgroup	it is used; the caller ID is chosen on every attempt by the subgroup's caller-ID policy
none	`422 dialing_route_unresolved`
more than one	`422 dialing_route_ambiguous`, with the candidates named in the message as `trunk_id/route_id` pairs

Ambiguity is never resolved by picking the first candidate: dialing from another trunk would spend another caller ID and another quota, and you would find out from the invoice. Read the candidates with:

```
GET /v1/dialing-routes?agent_id=collections-agent
```

The response is `{"dialing_routes": [{"trunk_id", "trunk_name", "route_id", "route_name", "agents", "caller_numbers"}]}` — no trunk connection settings or credentials. An empty `agent_id` lists every route visible to your organization.

> `trunk_id` **is the anchor.** Name it and the rest follows the platform's existing rules: the subgroup is resolved from the trunk and the agent at launch, and the caller ID is chosen by the subgroup itself, following its caller-ID policy. Naming `route_id` or `from_number` **without** `trunk_id` is refused with `422 dialing_binding_incomplete`: the platform will not guess which trunk you meant. A trunk named explicitly but unusable (agent not in its subgroup, empty pool) is not refused on creation — it becomes the `binding_invalid` blocker.

Attestation

Dialing requires a consent attestation: `{"attestation": {"confirmed": true}}`. The record is stored immutably against the version of the data it covers and the wording of the campaign's consent tier, and your API key is the signer — issuing the key is the human act the signature is attributed to (the platform records who issued the key and when).

A confirmation of one upload does **not** carry over to a new one; on `POST …/launch` the `attestation.import_id` field is therefore mandatory (→ `422 attestation_import_id_required`), and an `import_id` that is not the campaign's current import yields the `attestation_stale` blocker. Batches of added calls are signed separately.

Do-not-call matches: `dnc_screen`

`dnc_screen` is the authoritative result. It screens every readable source row before market rejection and deduplication, so `matched_rows` may be larger than `suppressed_contacts`.

Field	Type	Meaning
`status`	string	`clean` | `matches` | `screen_unavailable` (the import has no recorded source-row screening).
`checked_rows`	integer | null	Readable rows screened.
`unscreened_rows`	integer | null	Rows with an unreadable phone, not screened.
`matched_rows`	integer | null	Rows whose number is on the registry (before deduplication).
`suppressed_contacts`	integer | null	Distinct numbers matched.
`fingerprint`	string	Deterministic hash of the exact match composition; a `skip` decision is bound to it.
`items`	array	Up to 200 matches: `{"row_number", "phone", "phone_key", "contact_name", "dnc_entry_id", "source", "reason", "recorded_at"}`.
`has_more`, `hidden_rows`		Whether `items` is truncated and by how much.
`all_dialable_suppressed`	boolean	Every pending contact is matched.
`as_of_launch`	boolean	`false` — live against the registry; `true` — frozen snapshot taken at launch.
`screened_at`, `audit_id`	number | null, integer | null	When the frozen snapshot was taken and its audit record.
`decision`, `import_id`	string	Frozen snapshot only: `clean` | `skip`, and the import it covers.

`import.dnc_matches` is a compatibility projection of `items` (the same objects plus a `row` alias of `row_number`); `import.dnc_matched` equals `matched_rows` (`0` when the screen is unavailable).

Matches block launch with `dnc_matches`. Sending `dnc: "skip"` records the same audited decision a human makes in the cabinet. It applies to this exact `import_id` **and fingerprint**: a new match or a replacement registry entry makes the

decision stale, and launch-check/409 returns the refreshed screen. If every pending contact is matched, `all_contacts_suppressed` stays unconditional. `skip` with a clean screen records nothing and the launch proceeds normally.

The default `dnc: "block"` on an explicit launch is recorded as a decision too, and a later `block` revokes any earlier skip. An explicit launch that omits `dnc` therefore never rides on a skip somebody approved in the cabinet: pass `dnc: "skip"` yourself when that is what the call means. A decision needs an upload to apply to: `dnc` on a launch of a campaign without an import → `409 dnc_decision_without_upload`; on an import without a recorded projection → `409 dnc_screen_required`.

Before the first launch the screen and the matched-only report follow the live registry. The launch operation re-screens immediately before the transition and freezes all matched rows atomically with the import marker, attestation and campaign state. After launch, reads and reports use that snapshot (`as_of_launch: true`) even when the registry changes. Contacts matched mid-campaign are still suppressed at dial time; `progress` counts them separately.

Idempotency of campaign requests

`POST /v1/campaigns` and `POST /v1/campaigns/{id}/calls` share one protocol:

- The key is reserved **before** the work starts, atomically. A second request with the same key while the first is still running gets `409 idempotency_key_in_flight` — wait for the first answer instead of retrying again.
- The body is fingerprinted alongside the key (canonical JSON, key order irrelevant). The same key with a **different** body is `422 idempotency_key_reuse`, never a silent replay of the earlier answer.
- The same key with the same body replays the stored `201` byte for byte.
- Keys are scoped to your API key, the endpoint and — for `calls` — the campaign, and remembered for 24 hours.
- On creation, a refusal **before** the campaign exists frees the key; a refusal **after** it exists keeps it. On `calls`, any refusal frees the key.

TECHNICAL REFERENCE

Managing a campaign

Launch, pause, cancel, progress, statistics and adding batches of numbers.

Method and path	What it does
`GET /v1/campaigns`	list, newest first: `limit` (1–200, default 50) and `cursor` → `{campaigns, has_more, next_cursor}`. Items are short views: `campaign_id, status, name, agent_id, created_via, launched_via, starts_at, ends_at, timezone, timezone_source, accepts_call_additions, auto_speed, auto_rate_current, auto_rate_needed` — no blockers
`GET /v1/campaigns/{id}`	the campaign object (`import: null`) with live/frozen `dnc_screen` and the read-path `blockers` / `warnings`
`GET /v1/campaigns/{id}/launch-check`	`{blockers, warnings, dnc_screen}` — `dnc_screen` here is always the **live** screen
`POST /v1/campaigns/{id}/launch`	`{"attestation": {"confirmed": true, "import_id": "imp-…"}, "dnc": "block"}`. Success `200` with the campaign object (`blockers: []`); refusal `409 launch_blocked`
`POST /v1/campaigns/{id}/calls`	add a batch of numbers, see below
`POST /v1/campaigns/{id}/pause`	`running` → `paused`. Dialed calls play out; no new dials
`POST /v1/campaigns/{id}/resume`	`paused` → `running`. Re-runs the launch checks first — anything may have changed while paused
`POST /v1/campaigns/{id}/cancel`	any non-final status → `cancelled`. Irreversible; calls already dialed play out
`PATCH /v1/campaigns/{id}`	change the dialing speed or auto-speed mode: `{"calls_per_minute": N}`, `{"auto_speed": true}` or both, see below
`GET /v1/campaigns/{id}/progress`	contact states, holds, pace, do-not-call and frequency counters
`GET /v1/campaigns/{id}/rate-history`	the auto-speed recomputation log, see below

Statuses. `draft` → (launch) → `scheduled` (when `starts_at` is in the future) or `running`; `scheduled` becomes `running` when the window opens; `running` ⇄ `paused`; `running` → `completed` when the queue drains (nothing pending and nothing in progress) — **except** for a campaign with `accepts_call_additions: true`, which completes only when `ends_at` passes; any non-final status → `cancelled`. `completed` and `cancelled` are final. Reaching `ends_at` closes every campaign.

On `resume` the blockers `no_pending_contacts`, `attestation_required`, `attestation_stale`, `contacts_geo_rejected` and `dnc_screen_required` are not applied.

The `pause` / `resume` / `cancel` responses carry the campaign object **without** `blockers` and `warnings`: those paths do not run the launch checks, and an empty array would read as "checked and clean". Use `launch-check` when you need them.

Repeating a transition is not an error. A campaign already in the target state answers `200` with its current object — a retry after a network timeout is not a second intention. A transition impossible *from* the current state (resuming a cancelled campaign, pausing a draft) is `409 campaign_transition_invalid`. The difference between "already done" and "cannot be done" is the one you need.

A blocked launch or resume is an error, not a `200`. Creation answers `201` because a campaign and a report really were created; a `launch` — or a `resume`, which re-runs the same checks — that starts nothing answers `409 launch_blocked`, and the reasons travel in the same response:

```
{
  "error": {
    "type": "conflict_error",
    "code": "launch_blocked",
```

```
    "status": 409,
    "message": "The campaign did not start: error.blockers names every check that did not pass",
    "blockers": [
      {"code": "agent_not_published", "field": "agent_id"},
      {"code": "dnc_matches", "field": "contacts", "dnc_screen": {"status": "matches", "…": "…"}}
    ]
  }
}
```

The rule across the whole campaign surface: **an action that changed nothing is always an error; a `2xx` means the state in the body is the new state.**

Changing the dialing speed

`PATCH /v1/campaigns/{id}` with the body `{"calls_per_minute": N}` changes the speed of a campaign in `draft`, `scheduled`, `running` or `paused`. `N` is any whole number from 1 to 200, as on creation. The second body field is `auto_speed`: `true` turns on auto speed, `false` returns the campaign to `calls_per_minute`. Send one field or both; at least one is required. These are the only settings of an existing campaign you can change through `/v1`.

```
curl -s -X PATCH "$BASE_URL/v1/campaigns/cam-7f3a"   -H "Authorization: Bearer $CFLOW_API_KEY"   -H "Content-Type:
application/json"   -d '{"calls_per_minute": 30}'
```

- **When it takes effect.** A `running` campaign dials at the new speed from its next dialing cycle, without a restart; a `paused` one — from its resume; a `scheduled` one — from its start. A `draft` just stores the value.
- **Response.** `200` with the campaign object, as for `pause` / `resume`: without `blockers` and `warnings`.
- **Repeating the same value** answers `200` and changes nothing. The request is idempotent, so no `Idempotency-Key` is needed.
- **Audit.** Every real change is recorded with the old value, the new value and the campaign status, signed by your key.
- **Campaigns of the compatibility adapter.** The adapter sets the speed once, from the first request of the day. A change made here lasts until that day's campaign closes; the next day's campaign takes the speed from the adapter request again.

Status	`error.code`	Cause
404	`campaign_not_found`	No such campaign in your organization.
409	`campaign_closed`	The campaign is `completed` or `cancelled` — also when it closed while the request was in flight.
422	`invalid_request`	A field other than `calls_per_minute` and `auto_speed`, an empty body, a speed that is not a whole number, or an `auto_speed` that is not a boolean.
422	`campaign_spec_invalid`	A whole number outside 1–200.
422	`campaign_auto_speed_ends_at_required`	`auto_speed: true` on a campaign without `ends_at`.
422	`campaign_auto_speed_working_hours_required`	`auto_speed: true` on a campaign over several days without `working_hours`.

The speed can also be changed in the cabinet — see Dialing speed after launch.

Adding numbers to an existing campaign

```
POST /v1/campaigns/{id}/calls
Idempotency-Key: <unique string>        (required)

{
  "calls": [
    {"phone": "+77001234567", "timezone": "Asia/Almaty", "variables": {"debt": "15000"}},
    {"phone": "+77007654321", "name": "Daniyar"}
  ],
```

```
  "attestation": {"confirmed": true}
}
```

The campaign owns the agent, the dialing binding, the retry rules and the working hours; the batch carries **numbers, variables and a signature** only. Each element has the shape of one `planned_calls` row (Campaign fields) — its keys are column names. Unknown top-level keys are ignored: an `agent_id` in the body does not change the campaign's agent, and a `dnc` key does not bypass the registry.

A batch is accepted into any campaign that is not `completed` or `cancelled` — that is, into `draft`, `scheduled`, `running` and `paused`, whether or not it is open to call additions. A closed one answers `409 campaign_closed`.

Order of refusals, all before any row is parsed: `422 idempotency_key_required` → `422 calls_required` (empty `calls`) → `422 calls_batch_too_large` → `422 attestation_invalid` → `404 campaign_not_found` → `409 campaign_closed` → idempotency (`422 idempotency_key_reuse` / `409 idempotency_key_in_flight` / replay). Then the rows are parsed: `409 market_package_not_installed`, `422 contact_rows_rejected`, `502 contact_file_storage_unavailable` as on creation, and the two batch refusals below.

> **The batch is accepted whole or refused whole.** One row with a data error refuses the entire batch with `422 calls_rejected`; one number on the do-not-call registry refuses it with `422 calls_dnc_matched` — a distinct code, so your system can tell "fix the rows" from "remove these people from the list". When a batch has both kinds of problem, `calls_rejected` wins. Structural refusals of the parser (no phone column, no usable rows, …) are `calls_rejected` too. No contacts are created for a refused batch, and no signature is recorded.

Both refusals carry `error.import`:

```
{
  "error": {
    "type": "invalid_request_error",
    "code": "calls_rejected",
    "status": 422,
    "message": "…",
    "import": {
      "import_id": "imp-b87d984fe7",
      "status": "refused",
      "refusal_code": "calls_rejected",
      "rows_seen": 2,
      "accepted": 0,
      "warned": 0,
      "rejected": 1,
      "dnc_matched": 0,
      "issues_total": 1,
      "issues": [{"row_number": 2, "code": "phone_unreadable", "severity": "error"}]
    }
  }
}
```

Here `issues` holds the first 50 issues; the full per-row report is `GET /v1/imports/{import_id}/report` (and `?only=dnc_match` for the registry matches). A refused batch is kept as an import record with `status: "refused"` for exactly that purpose.

Success is `201`:

```
{
  "campaign": {
    "campaign_id": "cam-…", "status": "running", "name": "june collections",
    "agent_id": "collections-agent", "created_via": "api", "launched_via": "api",
    "starts_at": null, "ends_at": null, "timezone": "Asia/Almaty",
    "timezone_source": "request", "accepts_call_additions": true
  },
  "import": { "import_id": "imp-…", "status": "accepted", "rows_seen": 2, "accepted": 2, "warned": 0, "rejected":
```

```
0, "duplicates_found": 0, "dnc_matched": 0, "refusal_code": "", "issues": {}, "dnc_screen": {"status": "clean",
"as_of_launch": false, "…": "…"}, "dnc_matches": [] },
  "blockers": [{"code": "campaign_paused", "field": "status"}],
  "warnings": [{"code": "starts_immediately", "field": ""}, {"code": "frequency_policy_active", "field": ""}]
}
```


`campaign` is the short view (the same object as a list item). `blockers` means "accepted, but not dialing right now, and why": the same launch checks as `launch-check` minus the ones a fresh batch makes moot (`no_pending_contacts`, `attestation_required`, `dnc_screen_required`, `all_contacts_suppressed`, `dnc_matches`), plus `campaign_paused`. Items on this path carry `code` and `field` only.

Rules worth knowing before you stream numbers from a CRM:

- **Cap** — up to 1000 rows per request by default (a deployment setting): `422 calls_batch_too_large` above it, `422 calls_required` for an empty batch. One request counts once against the per-key rate limit.
- **Signature** — `attestation` must be exactly `{"confirmed": true}`: no other keys, no `import_id` (`422 attestation_invalid` otherwise). Every accepted batch is signed separately and bound to its own upload; the launch signature is not reused for rows that did not exist at launch.
- **Time zone** — in a campaign with `use_contact_timezone: true` under the Kazakhstan market profile, a row without a `timezone` is a data error (`timezone_required`) and refuses the batch: the platform will not fall back to the campaign zone for a row nobody looked at. File uploads and `planned_calls` on creation keep the fallback.
- **Duplicates** — inside a batch `duplicate_action` applies as it does for a file. A number that already has a contact in the campaign — even one already dialed — is **accepted again as a new contact and dialed again**: for an adapter that manages its own redials, a repeat *is* the redial. If your system manages redials, keep the default `retry_config.attempts = 1`; with a larger budget redials become two-level.
- **The batch never becomes the campaign's launch import** — the frozen do-not-call snapshot, the launch signature and the `dnc_screen` of `GET /v1/campaigns/{id}` keep pointing at the import the campaign was launched with. The batch's own screen is in the `201` body and at `GET /v1/imports/{import_id}`.
- **Idempotency** — the key is scoped to the campaign, so the same key and body sent to two campaigns are two batches. A refused batch **frees** the key: fix the rows and resend with the same key (a second refused import record is created — diagnostically useful, no calls involved). A repeat of an accepted batch replays the answer; the same key with a different body is `422 idempotency_key_reuse`.

Campaigns open to added numbers

An ordinary campaign completes itself as soon as its queue drains — and a completed campaign accepts no calls. To keep a campaign waiting for numbers, create it with `"accepts_call_additions": true`. Such a campaign:

- **requires** `ends_at` (`422 campaign_open_ends_at_required`), no further than the operator's horizon — 30 days by default (`422 campaign_open_ends_at_too_far`);
- may be created with no `planned_calls` at all and starts running empty (no `no_pending_contacts` blocker);
- is closed only by `ends_at` arriving or by `cancel`, never by a drained queue.

The flag is set at creation and cannot be changed later; it is returned in the campaign object and in the list.

Progress and contact states

`GET /v1/campaigns/{id}/progress`:


```
{
  "status": "running",
  "total": 120,
  "by_state": {"pending": 80, "in_progress": 3, "done": 30, "exhausted": 5, "suppressed": 2},
  "ready": 78,
  "holds": 2,
  "held_reasons": {"quiet_hours": 2},
  "suppressed_reasons": {"dnc_suppressed": 2},
  "dnc_suppressed_at_launch": 1,
```

```
  "dnc_suppressed_mid_campaign": 1,
  "backpressure": 0,
  "infra_retries": 0,
  "next_attempt_at": 1788942000.0,
  "frequency": {"held": 0, "blocked": 0, "by_governing_cap": {}},
  "rate": {"target_per_minute": 10, "actual_per_minute": 4}
}
```

`by_state` uses the platform's own state names — there is no translation layer, because the difference between "reached" and "out of attempts" matters to you. **States with a zero count are omitted.**

State	Meaning
`pending`	waiting to be dialed
`in_progress`	being dialed right now
`awaiting`	dialed; waiting for the conversation analysis verdict (`expert` strategy)
`done`	final — no further attempts: the call connected, or the call outcome rules out a retry (the number does not exist, blocked by a spam filter, in the US — the callee declined the call)
`exhausted`	attempts stopped without a conversation: the attempt budget or the infrastructure retries ran out, the next retry would fall after the campaign end, or the campaign ended while the contact was waiting (then it was never called)
`suppressed`	excluded before dialing (do-not-call, geography, policy)

`rate.target_per_minute` is the target pace, `rate.actual_per_minute` is how many dials went out in the last minute. On a campaign with auto speed, `rate` also carries:

Field	Meaning
`auto`	`true` — the campaign runs on auto speed.
`source`	`worker` — the dialer holds the target after a recomputation; `estimate` — an estimate before the first recomputation.
`remaining_dials`	How many dials are still needed, callbacks included.
`open_minutes`	Working minutes left until `ends_at` .
`cap_per_minute`	The calls-per-minute limit of the campaign's subgroup; `null` — no limit.
`capped`	`true` — finishing in time needs more than the limits allow: part of the base will not be reached before `ends_at` .
`concurrency_slots` , `in_flight` , `talking`	Under an organization concurrent call limit — how many calls are allowed, how many are in flight and how many of them are talking.
`ready_now`	How many open contacts can be dialed right now — the rest are still waiting out the retry interval.
`interval_bound`	`true` — the campaign dials below the target not because of the limits: there are not enough contacts to dial at the needed speed, the rest are waiting out the retry interval. The speed itself is not changed.

The other `rate` keys are internal inputs of the estimate, and their set may change.

`ready` is the number of pending contacts dialable right now; `holds` counts pending contacts held back (reasons in `held_reasons`); `suppressed_reasons` breaks down the `suppressed` state; `dnc_suppressed_at_launch` + `dnc_suppressed_mid_campaign` = `suppressed_reasons.dnc_suppressed` ; `frequency` reports contacts currently held or blocked by the per-number frequency caps and the caps governing them. Reason keys inside `held_reasons` , `suppressed_reasons` and `by_governing_cap` are machine codes and may grow.

> An ordinary campaign closes itself when its queue drains — a campaign created with `accepts_call_additions: true` does not (only `ends_at` or `cancel` closes that one). There is no "campaign finished" webhook either way — poll `progress` or `GET /v1/campaigns/{id}`.

Auto-speed recomputation log

`GET /v1/campaigns/{id}/rate-history` — one entry per auto-speed recomputation, oldest first:

```
{
  "campaign_id": "cam-7f3a",
  "entries": [
    {"id": 412, "recorded_at": 1793605200.0, "rate": 24, "estimate": {"…": "…"}}
  ]
}
```

`rate` is the pace the campaign dialed at after that recomputation; `estimate` holds the inputs it came from: dials still needed, working minutes, the callback share, the subgroup and concurrency ceilings, calls in flight. The content of `estimate` is internal and may change. Paging — `?after=<id of the last entry read>` and `?limit=` (default 1000, at most 5000). A campaign without auto speed returns an empty list; another organization's or a non-existent campaign — `404 campaign_not_found`.

Withdrawing a contact from dialing

Stop dialing a number before the call happens: the debt was paid, the number belongs to someone else, the case was settled in another channel.

Method and path	Body	Scope
`POST /v1/campaigns/{id}/contacts/withdraw`	`phones` **or** `contact_ids`	that campaign, in any status but `completed` and `cancelled` (a draft is allowed)
`POST /v1/contacts/withdraw`	`phones`	every **live** campaign of your organization — `scheduled`, `running`, `paused`

```
curl -s -X POST "$BASE_URL/v1/contacts/withdraw" \
  -H "Authorization: Bearer $CFLOW_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"phones": ["+7 701 123 45 67", "77015550011"]}'
```

`200 OK` lists the contacts by outcome; all five keys are always present:

```
{
  "operation_id": "wd-3f9c2a71b0",
  "withdrawn":         [{"contact_id": "cc-…", "campaign_id": "cam-…", "phone": "7011234567"}],
  "already_withdrawn": [],
  "not_withdrawable":  [{"contact_id": "cc-…", "campaign_id": "cam-…", "phone": "7015550011", "reason":
"in_call"}],
  "not_found": [],
  "invalid":   []
}
```

- **What is withdrawn.** A contact waiting to be dialed (`pending`, a scheduled redial included) or waiting for the analytics verdict (`awaiting`). It moves to `suppressed` with the reason `withdrawn_by_client` and is never dialed again in that campaign. For an `awaiting` contact the call **has already happened**: the withdrawal cancels the redial the verdict might have scheduled.
- **What is not.** `not_withdrawable` names the reason: `in_call` — the call is in progress or the contact was just taken for dialing (the withdrawal does not hang up; use `POST /v1/calls/{call_id}/end`, and withdraw again after the call if a

redial must not happen), `done`, `exhausted`, or `already_suppressed` — excluded earlier by the platform itself (do-not-call, a frequency cap, geography); that earlier reason is kept.

- **Repeating is safe.** No `Idempotency-Key` is needed: a contact you withdrew before comes back in `already_withdrawn`.
- **Addressing.** A number matches by its **last ten digits**, so any spelling works, and a number may be sent as a string or an integer. One number reaches every contact that carries it — in a campaign with duplicates, all of them. A contact id is the `metadata.contact_id` of the call webhooks; a contact that was never dialed is addressed by its number.
- **The organization endpoint reaches cabinet campaigns too**: every live campaign of your organization, however it was created. Drafts and closed campaigns are not touched.
- **Partial processing.** One bad address does not fail the request: `invalid` returns what could not be read (not a number, no digits, an empty id), `not_found` what found no contact — both exactly as you sent them. At most 1000 addresses per request by default.
- **Consequences.** Withdrawing the last waiting contacts of a running campaign completes it on the next dialer pass, unless the campaign is open to added numbers. A withdrawn number returns to dialing only with a new batch of calls.

> There is no withdrawal screen in the cabinet yet: the `withdrawn_by_client` reason shows up in the campaign progress breakdown, but the operation itself is available only through the API.

TECHNICAL REFERENCE

The upload report

What came back for each row of the file, and the row-level issue codes.

Method and path	What it returns
`GET /v1/imports/{import_id}`	the import summary (the same object as `import` in the creation response) plus live/frozen `dnc_screen`
`GET /v1/imports/{import_id}/report?format=csv`	per-row report: `csv` (default) or `xlsx` (anything else → `422 report_format_invalid`); filename `import-<id>-report.<ext>`
`GET /v1/imports/{import_id}/report?only=dnc_match&format=csv`	only the exact do-not-call matches; filename `import-<id>-dnc.<ext>`. Any other `only` → `422 report_filter_invalid`; an import without a recorded projection → `409 dnc_screen_required`

Imports are visible only through campaigns of your organization (`404 import_not_found` otherwise).

The import summary object

Field	Type	Meaning
`import_id`	string	
`status`	string	`accepted` | `refused`.
`rows_seen`, `accepted`, `warned`, `rejected`, `duplicates_found`	integer	Row counters. On a refused batch of added calls `rejected` / `warned` are counted from the issues.
`dnc_matched`	integer	`matched_rows` of the screen (`0` when unavailable).
`refusal_code`	string	`""` for an accepted import; otherwise one of `unreadable_file`, `no_rows`, `no_phone_column`, `ambiguous_phone_column`, `too_many_rows`, `no_usable_rows`, `missing_required_column`, `calls_rejected`, `calls_dnc_matched` (file-only codes such as `unsupported_format`, `file_too_large` do not occur on `/v1`).
`issues`	object	**Keyed by issue code**: `{"<code>": {"severity": "error"|"warning", "count": n, "rows": [row numbers, first 10]}}`. Empty object when there are no issues.
`dnc_screen`, `dnc_matches`		See Campaign fields.

> `issues` in the summary is an **object keyed by code**, not an array. The array form (`{"row_number", "code", "severity"}` items, first 50) appears only inside `error.import` on a refused batch.

Row-level issue codes

The `code` column of the report, and the keys of `issues`:

Code	Severity	Meaning
`phone_unreadable`	error	No readable phone (fewer than ten digits).
`row_empty`	error	Empty row.
`duplicate_row`	error	Duplicate number under `duplicate_action: "reject"`.
`duplicate_ignored`	warning	Duplicate skipped or replaced (`keep_first` / `keep_last`).

Code	Severity	Meaning
`timezone_invalid`	warning	Unknown `timezone` value (the campaign zone is used).
`timezone_required`	error	`calls` batch only: no `timezone` in a contact-zone campaign under the kz profile.
`group_role_invalid`, `group_key_missing`, `group_key_too_long`	error	Contract-group columns inconsistent.
`variable_name_collision`	warning	Two columns map to one variable.
`dnc_match`	warning	The number is on the do-not-call registry.
`state_mismatch`, `state_invalid`, `zip_invalid`, `city_mismatch`, `timezone_override`, `state_only`, `multi_zone`	warning	US market package: location resolution flags.
`unresolvable_location`, `non_callable_geography`	error	US market package: the row cannot be dialed.

The report file

The report columns are `row_number`, `severity`, `code`, `supplied`, `duplicate_of_row` (`supplied` is a JSON object with the submitted values). There is **no message column**: the machine-readable contract of the report is the `code`, and rendering it as human text is the consumer's job.

The summary block at the end of the file (`import_summary`) uses the field keys as labels — `filename, format, encoding, phone_column, rows_seen, accepted, warned, rejected, duplicates_found, status`, plus `refusal` with the refusal code for a refused import; for `planned_calls` and `calls` sources `filename` and `encoding` are empty and `format` is `json`. The `dnc_match` report has the columns `row_number, phone, contact_name, phone_key, dnc_entry_id, source, reason, recorded_at` and a `dnc_screen_summary` block (`checked_rows, unscreened_rows, matched_rows, suppressed_contacts`).

Two conventions of the shared upload machinery apply on every path alike — a file in the cabinet, the `planned_calls` of campaign creation, and an added `calls` batch:

- **row numbers count the header as row 1**, so the first planned call is `row_number` 2 and the N-th is N+1 — in `issues`, in `error.import` and in the report file;
- **the CSV is UTF-8 with a BOM and CRLF line endings** (so Excel opens Cyrillic correctly): decode it as `utf-8-sig`, or the first header cell reads as `row_number`.

Summary rows carry field keys instead of a numeric `row_number` and are not data rows.

TECHNICAL REFERENCE

Individual calls

`POST /v1/calls/phone`, the call object, reading, ending and listing calls.

`POST /v1/calls/phone`

One outbound call: the platform registers the call, answers immediately, and dials the destination from a durable dial queue through the specified trunk, connecting the published agent when the callee answers.

Field	Type	Required	Default	Description
`agent_id`	string	yes	—	Agent name. Must be **published** and listed in the `agents` of the outbound subgroup being used.
`trunk_id`	string	yes	—	The trunk to dial through.
`to_number`	string	yes	—	Destination number; normalized before dialing.
`route_id`	string	no	`""`	Outbound subgroup of the trunk. Empty → resolved from (`trunk_id` , `agent_id`): the agent attached to exactly one subgroup of the trunk selects it; several → `409 agent_subgroup_ambiguous` ; none → `403 agent_not_in_subgroup` . Unknown `route_id` → `404 subgroup_not_found` .
`from_number`	string	no	`""`	Caller ID; must be one of the **subgroup's** `caller_numbers` . An empty string leaves the choice to the subgroup, following its caller-ID policy: `random` (the default) picks one at random, `round_robin` walks the pool in turn. Candidates are the subgroup pool intersected with the organization grant pool; when a single number remains, that is the one used. A subgroup with an empty pool refuses every value, including empty (`caller_number_not_allowed`); an empty intersection with the grant gives `caller_number_outside_grant_pool` .
`dynamic_variables`	object	no	`{}`	Variables handed to the agent's flow (customer name, account data).
`metadata`	object	no	`{}`	Your opaque correlation payload. Echoed back unchanged in the call object and in **every** webhook — put your lead ID here.
`ring_duration`	number | null	no	`null`	Per-call ring limit in seconds; must be > 0. `null` uses the agent's default.

Number normalization. All non-digit characters are stripped from `to_number` (`"+1 (555) 123-4567"` → `"15551234567"`). A number with no digits is rejected with `400` . If the **subgroup** defines a dial prefix, it is prepended to the normalized number; allowed destinations are checked against the normalized number **before** the prefix.

Order of checks. The dial-queue ceiling (`429 dial_queue_full`) is checked first, before any validation. Then, synchronously and before the call is registered: a dialing stop by the platform (409), trunk and grant (404), subgroup and agent attachment (404/409/403), caller ID (400), destination policy (403), subgroup limits (429), `ring_duration` (400), and finally the published version of the agent (409). A refusal at any of these steps leaves no call behind. Because the limits are checked before the publication check, a 429 does not tell you whether the agent is published.

The `Idempotency-Key` header is recommended here (and effectively required after a `429`): a repeat with the same key returns the **already-created call** instead of dialing a second time. Keys are remembered for 24 hours and are scoped to your API key and endpoint. The replay returns the call object **as it was at creation** (`status: "queued"`), not its current state — poll `GET /v1/calls/{call_id}` for that.

> **Success is `202 Accepted` , not `201` .** The call is registered and queued for dialing; the dial itself happens asynchronously after the response. Store `call_id` : it is the correlation key for polling and for every webhook.

Call initiation errors

Status	error.code	Condition
400	`caller_number_not_allowed`	`from_number` is not one of the subgroup's numbers
400	`caller_number_outside_grant_pool`	`from_number` is outside the pool granted to your organization (trunks used under a grant only)
400	`destination_number_invalid`	`to_number` contains no digits
400	`ring_duration_invalid`	`ring_duration` ≤ 0
403	`agent_not_in_subgroup`	the agent is not attached to the subgroup (this is also the answer for an agent name that does not exist)
403	`destination_not_allowed`	the destination is blocked by the subgroup's policy
404	`trunk_not_found` / `subgroup_not_found`	unknown `trunk_id` (or no grant for it) / unknown `route_id`
409	`agent_subgroup_ambiguous`	the agent is in several subgroups of the trunk — pass `route_id`
409	`agent_not_published`	the agent has no published version
409	`dialing_stopped`	outbound dialing for your organization is stopped by the platform. Calls queued before the stop are not lost: they wait until the stop is lifted and are dialed after that. Only the platform can lift it — contact support
422	`invalid_request`	the body fails schema validation; field-level details are in `message`
502	`dial_failed`	only when the operator has disabled the dial queue (synchronous fallback). In the default asynchronous mode a dial failure is **not** an HTTP error — the call is created (`202`) and later reads as `status: "failed"` , `status_reason: "dial_failed"`

The call object

Returned by `POST /v1/calls/phone` , `GET /v1/calls/{call_id}` and `POST /v1/calls/{call_id}/end` .

Field	Type	Description
`call_id`	string	Call identifier: a sign plus a full UUID — `call-out-<uuid>` outbound, `call-in-<uuid>` inbound, `web-<uuid>` web. The UUID is 36 lowercase characters with dashes, for example `call-out-0f8fad5b-d9cb-469f-a165-70867728950e` . It carries no phone number or other data.
`agent_id`	string	Agent name.
`flow_version`	string | number | null	Version marker of the published flow the call ran on. **Its type is not stable**: while the call is live it is an opaque fingerprint string; once the call is stored it is the integer version number. Treat it as opaque and use `flow_version_id` for identity.
`flow_version_id`	number | null	Numeric ID of the published flow version.
`channel`	string	`phone` | `web` .
`direction`	string	`outbound` | `inbound` | `web` .
`status`	string	Live state: `queued` → `ringing` → `ongoing` → `ended` | `failed` .
`call_status`	string	Final bucket: `completed` | `not_connected` | `error` ; `""` until finalized.
`status_reason`	string	Fine-grained reason within the bucket; `""` until finalized.
`outcome_category`	string	The row of the campaign's outcome breakdown the call falls into: `talked` (Human Conversation), `transferred` , `screener_human_handoff` , `voicemail` , `screener_voicemail_handoff` , `screener_no_handoff` , `connected_no_conversation` and others; `""` until finalized. Also in the `GET /v1/calls` list items.

Field	Type	Description
`error_code`	string | null	Carrier / telephony error code when the platform received one; `null` otherwise. Diagnostic.
`message_left`	boolean	Answering-machine detection triggered and the agent left a message — spoken in full.
`callee_reached`	boolean	The dial reached the callee's phone: it rang or played early audio. For a call that did not connect, it tells "rang, not answered" from "never rang". `false` for inbound and web calls.
`voicemail_decision`	object | null	The voicemail decision of a campaign call: `mode` — the policy mode (`always` | `rules` | `never`), `source` — where the policy came from (`campaign_override` | `organization_policy` | `unconfigured`), `policy` — the policy itself, `blocked_by` — which condition withheld the message (`""` if none), `counters` — `voicemail_today` and `voicemail_7d` for the number, `outcome` — `left` (message left) | `not_left` (voicemail, the message was not spoken in full) | `suppressed` (the policy withheld it) | `no_voicemail` (no voicemail), `message` — how the message itself ended: `left` (played to the end) | `cut` (cut off midway) | `unvoiced` (never played) | `closed` (the machine closed before the message) | `error` (the message failed), `null` if the agent did not try to leave one; the counters count messages left only. `null` for a call outside a campaign, including one started through `/v1` . See Voicemail message policy.
`screener`	object | null	The call screener phase: `detected` — `true` ; `type` — which screener (`ios` | `assistant` | `google` | `cloaked` | `unknown`); `outcome` — how the phase ended (`connected` — put a person through, `message` — offered to take a message, `declined` — declined, `timeout` — ran out of time); `replies` — how many times the agent answered the screener; `hold_s` — how many seconds the screener held the line; `source` — how the screener was recognized (`phrase` | `model` | `watch`); `greeting` — the screener's first words. `null` if the call met no screener.
`started_at`	number | null	Start of the attempt (for phone calls — before dialing).
`answered_at`	number | null	The moment the callee answered; `null` for unanswered and web calls.
`media_started_at`	number | null	When audio media started flowing. Diagnostic.
`talk_from` , `talk_to`	number | null	Bounds of the talk-time interval used for `duration` . Diagnostic.
`ended_at`	number | null	End of the call; `null` while live.
`duration`	number | null	Talk time in seconds from `answered_at` . `null` for unanswered calls, for `error/not_finalized` , and while ringing; for a live call, the elapsed talk time.
`transcript`	array	Ordered turns: `{"role": "user"|"assistant", "text": string, "interrupted"?: true, "out_of_turn"?: true, "wait_request"?: true, "voicemail_greeting"?: true, "ts"?: number, "spoke_at"?: number}` . Only what the caller actually heard is included: an agent reply that never played (the call ended or it was cut before its first sound) is **left out** of the transcript entirely; you can see it only in the call card with **Show system events**, marked as not played. Since 14.10.2026 this also applies to old calls when a webhook is retried. A reply cut off mid-sentence stays, with the truncated text and `interrupted: true` . `voicemail_greeting: true` (caller lines only) — the answering machine greeting by which the call was detected as voicemail; the agent did not reply to it. `out_of_turn: true` (caller lines only) — the caller spoke over an agent reply that could not be interrupted; the line was kept outside the turn and is not an answer to the agent's next question. `wait_request: true` (caller lines only) — the caller asked the agent to wait and the agent paused (pause on client request); such a line also carries `out_of_turn: true` . These keys were added without a version bump: parsers that do not know them keep working.
`dynamic_variables`	object	Current conversation variables: your initial values plus what was collected during the call. On an **unanswered** call it holds the values you supplied for the contact — an empty object is not a no-answer marker, read `call_status` for that.

Field	Type	Description
`metadata`	object	Your payload, unchanged. A campaign call carries the campaign and contact keys instead — see Campaign calls.
`analysis_status`	string	`""` (not started) | `pending` | `processing` | `done` | `skipped` | `failed` | `interrupted`. Terminal: `done`, `skipped`, `failed`, `interrupted`.
`call_analysis`	object | null	Only when `analysis_status` is `done`: `summary`, `sentiment`, `custom_fields`, `call_successful`.
`health`	string | null	Technical call health: `ok` | `warning` | `error`; `null` until finalized. It reflects provider and infrastructure issues, **not** conversation quality.
`health_reasons`	array	Reason codes behind the verdict, see the glossary below.
`events`	array	Call events without details: `{"type", "turn", "ts", "title"?, "status"?}` (observer incidents, transfers and similar moments). Additive, diagnostic.
`cost`	object | null	`{"total_microusd", "analytics_microusd"}` — `null` in the responses your key receives.
`recording_url`	string	Stable download URL once the recording is ready; `""` before that. Request it with the API key of the call's organization (`Authorization: Bearer <API key>`).
`recording`	object | null	Recording metadata: `{"id", "status", "channels", "format", "duration", "size", "expires_at", "client_audio_lost_sec"}`. `status` — `recording` | `processing` | `ready` | `failed` | `deleted`; `channels` — `stereo` | `mono`; `client_audio_lost_sec` — seconds mid-call without the client's sound in the recording, `null` if not measured (see Webhooks).
`from_number`, `to_number`, `trunk_id`, `route_id`	string	Phone calls only.

Status lifecycle: `queued` (dialing initiated, no ring yet) → `ringing` → `ongoing` (callee answered) → `ended` (the conversation took place, including voicemail drops and transfers) or `failed`.

> `failed` **does not by itself mean the callee never answered.** Usually it comes with `call_status: "not_connected"` (`no_answer`, `busy`, `dial_failed`, …), but a platform failure also reads as `failed` with `call_status: "error"` (`internal_error`, `not_finalized`, `agent_unavailable`, `no_media`). Always read `call_status` / `status_reason`.

Web calls skip `ringing` and `failed`: they go `queued` → `ongoing` (first turn of history) → `ended`.

Asynchronous fields fill in after the call ends: `call_analysis` when post-call analysis completes, `recording_url` when the recording is finalized. The `call_analyzed` webhook notifies you when both are done — polling is a fallback, not the primary mechanism.

Reading and ending a call

`GET /v1/calls/{call_id}` returns the full call object; `404 call_not_found` for unknown IDs.

Calls recorded before the UUID format. Their old ID had a short tail (`call-out-<16 hex>`, `web-<16 hex>`), and an inbound one was built around the caller's number. Such calls now answer with an alias in the new format (`<sign><uuid>`, same sign) as `call_id` — in every response and webhook — while the old ID is still accepted on the way in: `GET /v1/calls/{old ID}` returns `200` with `call_id` equal to the alias. If you compare a response's `call_id` with an old stored ID as strings, expect the difference for those calls only; the `event_id` of their webhooks is unchanged.

`POST /v1/calls/{call_id}/end` hangs up a live call — a phone call still waiting in the dial queue is cancelled before dialing — and returns the call object.

> `end` is idempotent only **within the live-registry window**: calling it on a call that ended less than 5 minutes ago returns the object without an error; after that the registry entry is gone and the endpoint answers `404 call_not_found`, even though `GET /v1/calls/{call_id}` still returns the stored object. Do not use `end` as a way to read state.

`GET /v1/calls/{call_id}/turns` returns per-turn latency metrics of a finished phone or web call: an array of `{"turn", "metrics": {"vad_ms", "asr_ms", "eou_wait_ms", "classifier_ms", "llm_ms", "tts_ms", "turn_ms", …, "estimated"?: true}, "interrupted"?: true, "merged_from"?: number}`. A diagnostic surface for load testing; `404` for unknown IDs and for non-call sessions.

`merged_from` appears on a caller turn whose sentence was split by a pause and glued back together: it is the number of fragments, and the turn's phases are those of the first one. The key is additive: a client that ignores it parses the response the ordinary way. There is no `llm_first_sentence_ms` metric in the response.

`GET /v1/calls` — list calls

Parameter	Notes
`agent_id`	filter by agent
`direction`	`outbound` | `inbound` | `web`. **Not validated**: an unknown value yields an empty list, not `400`
`status`	only `ongoing` or `ended` (anything else → `400 invalid_filter_value`); `ended` includes failed calls
`call_status`	`completed` | `not_connected` | `error` (anything else → `400 invalid_filter_value`)
`status_reason`	exact match, e.g. `agent_hangup` (unknown value → `400 invalid_filter_value`)
`health`	`ok` | `warning` | `error` — one-click "problem calls only": `health=error`
`health_reason`	exact code match. Not validated: an unknown code yields an empty list
`from_number`, `to_number`	⚠ applied **after** pagination, to the current page only
`since`, `until`	bounds on call start, epoch seconds, inclusive
`limit`	1–200, default 50; out of range → `422 invalid_request`
`cursor`	cursor from a previous response; malformed → `400 invalid_cursor`

Response: `{calls: [...], has_more, next_cursor}`. List items are summaries without a transcript; fetch `GET /v1/calls/{call_id}` for the full object. Iterate by passing `next_cursor` back as `cursor` until `has_more` is `false`. Only phone and web calls are listed; editor test sessions and chats are not.

> `from_number` / `to_number` are page-local filters: a page may contain fewer than `limit` items (possibly zero) while `has_more` is still `true`. Do not treat an empty page as the end of results. For reconciliation by number, prefer filtering on your own `metadata` client-side.

TECHNICAL REFERENCE

Calls from your website: Web SDK

`POST /v1/calls/web`, the `@hubtalk-ai/web-sdk` browser SDK and the ready-made widget: a conversation with the agent right on your website.

A visitor to your website presses a button and talks to the agent from the browser — over the microphone, with no phone line. This is what the `POST /v1/calls/web` endpoint and the `@hubtalk-ai/web-sdk` browser SDK are for; the SDK also ships a ready-made widget: a button in the corner of the page and a call panel.

If someone just needs to talk to the agent and nothing has to be embedded, a demo link is enough. The Web SDK is for when the call starts on your website, on your domain, with variables your system knows.

How it works

1. The visitor presses the call button on your page.
2. The page asks **your server** to start a call. The server calls `POST /v1/calls/web` with the organization's API key and gets a session: `access_token` and `url`.
3. The server hands the session to the page, and the page passes it to the SDK. From there the conversation runs directly between the browser and the platform, and the call state reaches the page as events.

> **The API key never reaches the browser.** Only your server knows it. The page gets a one-time session for one call — it lets the browser into this conversation and nothing else.

What you need first

- **The agent is published.** The published version takes the call; a draft does not connect. A web call needs no telephony binding.
- **An API key is issued** — the **API keys** section, see Access key. The key stays on your server.
- **The page is served over HTTPS.** Without a secure connection the browser does not grant the microphone. `localhost` during development is the exception.

`POST /v1/calls/web`

Creates a web call and issues a session for the browser. Your server makes the request, with the `Authorization: Bearer <API key>` header.

Field	Type	Required	Default	Description
`agent_id`	string	yes	—	The agent name. Must be **published**.
`dynamic_variables`	object	no	`{}`	Scenario variables — the same `{{var}}` as in nodes: client name, order number.
`metadata`	object	no	`{}`	Your opaque payload for matching. Returned unchanged in the call object and in every webhook.
`avatar`	boolean	no	`false`	Ask for a live avatar on this call, see Live avatar. Only `true` or `false`: a string or a number is `422`.
`avatar_quality`	string	no	agent setting	Avatar video quality: `low`, `medium`, `high`, `very_high`. An unknown value does not break the call: the platform skips it and uses the agent's quality setting.

```
curl -X POST "$BASE_URL/v1/calls/web" \
  -H "Authorization: Bearer $HUBTALK_API_KEY" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: 6f1c2a90-site-call-1" \
```

```
  -d '{
    "agent_id": "clinic_booking",
    "dynamic_variables": {"client_name": "Emily"},
    "metadata": {"lead_id": "L-1042"}
  }'
```

Response — `201 Created`: the call object plus three session fields.

Field	Description
`access_token`	The token for the browser. Lives **2 hours** and is never refreshed.
`url`	The address the browser connects to.
`room`	The call room; equal to `call_id`.
`call_id`	The call id, `web-<uuid>`. Read the call with `GET /v1/calls/{call_id}` and find it in webhooks by this id.

You can hand the whole response to the page and pass it to the SDK as is: the SDK ignores extra fields.

Errors

Status	`error.code`	Condition
404	`agent_not_found`	no agent with this name in your organization
409	`agent_not_published`	the agent has no published version
422	`invalid_request`	the body fails schema validation; field details are in `message`
429	`org_concurrency_limit`	the organization's concurrent call limit is reached
503	`agent_dispatch_failed`	the platform has nowhere to connect the agent right now — retry later

Key and rate limit errors common to all endpoints are in Errors.

Retries. The `Idempotency-Key` header protects against a double call on retries: a repeat with the same key within 24 hours returns the same session, and no second call is made. One retry key per call: the next visitor needs a new one.

One session per call. The token lives 2 hours and is never refreshed, so issue a new session for every call rather than holding one on an open page.

The call in History and webhooks

A web call shows up in History right away as an active conversation on the **Web call** channel (`channel: "web"`, `direction: "web"`). Transcript, recording, post-call analysis and webhooks work as for a phone call. In the scenario, `{{direction}}` equals `web` — the agent can use it to talk to a website visitor differently than on the phone.

The lifecycle is shorter than a phone call's: `queued` → `ongoing` → `ended`, with no `ringing` and no `failed`. Number and trunk fields of a web call are empty, `answered_at` is `null`.

Installing the SDK

```
npm install @hubtalk-ai/web-sdk
```

```
import { VoiceClient } from '@hubtalk-ai/web-sdk';
```

Without a bundler. The package's `dist` folder holds ready-made files for a plain `<script>` tag: `voice-client.bundle.js` — the client, `widget.bundle.js` and `widget.css` — the widget. Host them on your own server:

```
<script src="/sdk/voice-client.bundle.js"></script>
<script>
```

```
  const client = new VoiceClient();
</script>
```

No third-party CDN is needed: everything the SDK requires is inside these files.

Your first call

On the server, an endpoint that calls `POST /v1/calls/web` and returns the response to the page. An example in Node.js (Express):

```
app.post('/api/start-voice-call', async (req, res) => {
  const r = await fetch(`${process.env.HUBTALK_BASE_URL}/v1/calls/web`, {
    method: 'POST',
    headers: {
      Authorization: `Bearer ${process.env.HUBTALK_API_KEY}`,
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      agent_id: 'clinic_booking',
      dynamic_variables: { client_name: req.user?.name ?? '' },
    }),
  });
  res.status(r.status).json(await r.json());
});
```

The server sets the call variables. If a value comes from the page, accept only the fields you expect yourself — that way a visitor cannot slip foreign data into the scenario.

On the page:

```
const client = new VoiceClient();

startButton.addEventListener('click', async () => {
  const session = await fetch('/api/start-voice-call', { method: 'POST' }).then((r) => r.json());
  await client.startCall(session);
});

client.on('transcript', ({ role, text, final, id }) => renderBubble(id, role, text, final));
client.on('callEnded', ({ reason }) => console.log('call ended:', reason));

endButton.addEventListener('click', () => client.stopCall());
```

> **Call `startCall` inside a click handler.** The browser allows the agent's audio and the microphone only after a user action. On iPhone Safari a network request between the click and `startCall()` can use up that action — in that case fetch the session ahead of time, for example when the call panel opens.

VoiceClient

Options

`new VoiceClient(options?)` — every option is optional.

Option	Default	Description
`audioContainer`	a hidden `<div>` the SDK creates	Where the SDK puts the `<audio>` elements with the agent's voice.
`logger`	`console`	Your own log sink `{ error, debug }`. Tokens are masked in logs either way.
`logLevel`	`'error'`	`'silent'`, `'error'` or `'debug'`.
`connectTimeoutMs`	15 seconds	How long to wait for the connection.

Methods

Method	What it does
`startCall(session, options?)`	Connects and turns on the microphone. Takes the `POST /v1/calls/web` response as is, or `{ accessToken, url }`. A second call during a call fails with `ALREADY_IN_CALL`. `{ audio: false }` — a text-only call.
`stopCall()`	Ends the call. With no call, does nothing and raises no error.
`sendText(text)`	Sends the visitor's text to the agent: the agent interrupts its speech and answers it as it would answer speech. Outside a call — `NOT_IN_CALL`; text over 1000 characters is truncated. The message is not added to the chat by itself — it comes back as a `transcript` event with `role: 'user'`.
`setMicrophoneEnabled(enabled)`	Turns the microphone off and on during a voice call; the call goes on. `isMicrophoneEnabled()` returns the current state.
`resumeAudio()`	Turns the agent's audio back on when the browser blocked it (the `audioPlaybackBlocked` event). Call it from an Enable sound button.
`getState()`	Connection state: `idle`, `connecting`, `connected`, `reconnecting`, `disconnected`, `error`.
`getAgentState()`	What the agent is doing: `unknown`, `initializing`, `listening`, `thinking`, `speaking`.
`getCallId()`	The call's `call_id` — the same one `GET /v1/calls/{call_id}` takes; `null` before connecting.
`on` / `off` / `once`	Subscribe to an event, unsubscribe, subscribe once.
`destroy()`	Ends the call and releases everything; the client is not usable afterwards.

Events

Event	Payload
`callStarted`	`{ callId }`
`callEnded`	`{ callId, reason }`. `reason`: `local` — you ended it (`stopCall`), `remote` — the agent ended it, `connection_lost` — the connection dropped, `error` — a failure.
`connectionStateChanged`	`{ state, previous }`
`agentStateChanged`	`{ state, previous }`
`agentSpeakingStarted` / `agentSpeakingStopped`	`{}`
`transcript`	`{ role, text, final, id }` — a line of the conversation, `role`: `user` or `agent`. While `final: false` the line is still growing: render one bubble per `id` and update it.
`microphoneStateChanged`	`{ enabled }` — the microphone is on or off.
`audioPlaybackBlocked`	`{}` — the browser blocked the agent's audio. The call goes on; show a button that calls `resumeAudio()`.
`error`	`VoiceError` — see below.

Text-only calls

`startCall(session, { audio: false })` starts the same call with no microphone and no sound: the browser asks for no permissions, and the conversation runs in text through `sendText()` and `transcript` events. This is the mode for a visitor who cannot or does not want to speak aloud. The agent's replies arrive at speaking pace.

SDK errors

Every failure is a `VoiceError` object: `{ code, message, state, cause? }`. Decide by `code`; `message` is a line for logs, and the visitor-facing text is yours.

Code	When
`INVALID_PARAMS`	the session has no `access_token` or `url`
`WEBRTC_UNSUPPORTED`	the browser does not support calls, or the page is not on HTTPS
`MICROPHONE_PERMISSION_DENIED`	the visitor denied the microphone
`MICROPHONE_UNAVAILABLE`	there is no microphone, or it is busy
`AUDIO_PLAYBACK_BLOCKED`	the browser blocked the audio again after `resumeAudio()`
`TOKEN_EXPIRED`	the session expired before the call started
`TOKEN_INVALID`	the platform rejected the session token
`CONNECTION_FAILED`	could not connect
`CONNECTION_LOST`	the connection dropped and did not come back
`ALREADY_IN_CALL`	a call is already running
`NOT_IN_CALL`	an action outside a call: `sendText`, a microphone toggle
`SERVER_ERROR`	any other failure

The ready-made widget

If you do not need your own layout, the package has a widget: a button in the corner of the page and a call panel with the agent's state, a timer, microphone and end buttons, a chat and the transcript after the call. Widget texts are in English or Russian.

```
import { mountWidget } from '@hubtalk-ai/web-sdk/widget';
import '@hubtalk-ai/web-sdk/widget.css';

const widget = mountWidget({
  container: document.getElementById('support'),
  locale: 'en',
  async createSession() {
    const r = await fetch('/api/start-voice-call', { method: 'POST' });
    if (!r.ok) throw new Error('Session unavailable');
    return r.json(); // { access_token, url, … } — never the API key
  },
});

// when leaving the page or removing the component
widget.destroy();
```

Without a bundler — the same files from `dist`:

```
<link rel="stylesheet" href="/sdk/widget.css">
<div id="support"></div>
<script src="/sdk/widget.bundle.js"></script>
<script>
  HubTalkWidget.mountWidget({
    container: document.getElementById('support'),
    createSession: async () => {
      const r = await fetch('/api/start-voice-call', { method: 'POST' });
      if (!r.ok) throw new Error('Session unavailable');
      return r.json();
    },
  });
</script>
```

Option	Default	Description
`container`	required	The page element the widget lives in.
`createSession`	required	A function that asks your server to start a call and returns the `POST /v1/calls/web` response. The widget calls it when the visitor presses **Start conversation**.
`locale`	`'en'`	Language of the widget texts: `'en'` or `'ru'`.
`mode`	`'standard'`	`'standard'` — the call and chat panel; `'avatar'` — the panel with the live avatar card.
`screenAdaptation`	`{ mode: 'height' }`	How the open panel sits on the screen. `'height'` — a corner panel half the screen tall. `'full-height'` — a panel from the site header down to the bottom of the screen; the widget finds the header itself, or pass its selector in `headerSelector`. On phones the panel fills the screen in either mode.
`renderCallDetails`	none	A function that draws your own form in the panel once — for example, a choice of request topic. `createSession` reads the form values itself. The widget hides the form during the call.
`avatar`	none	Avatar card settings for `'avatar'` mode, see below.

The widget styles come in a separate `widget.css` file and apply only inside the widget — they do not change the look of your page.

Live avatar

In `'avatar'` mode the panel shows **live video of an AI avatar**: a face that speaks with the agent's voice, in sync with the speech. The video travels over the same connection as the audio.

- **Platform support turns the avatar on for the agent** and sets which face appears in the call. Write to support naming the agent that needs the avatar, and ask which face was chosen.
- **Your server asks for the avatar** with `"avatar": true` in `POST /v1/calls/web`. The widget tells you when: `createSession` receives `{ avatar, avatarQuality }`, and `avatar` is `true` in `'avatar'` mode. This is a request, not a guarantee: if the agent has no avatar turned on, the call is audio-only, and the API response does not change.
- **The placeholder is yours.** While there is no video — before the call, in the first seconds, when the video drops — the stage shows your image `avatar.posterUrl` or a pair of silent clips. Pick a placeholder that looks like the avatar's face, or the switch will be noticeable.

```
mountWidget({
  container: document.getElementById('support'),
  mode: 'avatar',
  async createSession({ avatar, avatarQuality }) {
    const r = await fetch('/api/start-voice-call', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ avatar, avatar_quality: avatarQuality }),
    });
    if (!r.ok) throw new Error('Session unavailable');
    return r.json();
  },
  avatar: {
    name: 'Alina',
    subtitle: 'Clinic assistant',
    posterUrl: '/presenter/poster.jpg',
    liveCaptions: true,
  },
});
```

`avatar` option	Default	Description
`posterUrl`	required	The placeholder image, from your server.

avatar option	Default	Description
`listeningVideoUrl`, `speakingVideoUrl`	none	Silent "listening" and "speaking" placeholder clips — both or neither. Without them the image stays.
`name`, `subtitle`	"AI assistant", none	Name and caption in the panel header.
`quality`	agent setting	`low`, `medium`, `high`, `very_high`. Passed to `createSession` as `avatarQuality` — forward it as `avatar_quality`.
`liveCaptions`	`false`	The agent's last replies as captions under the video.

While the video plays, an **AI avatar** label shows on it. If your jurisdiction requires telling visitors they are talking to AI, that stays on your side: the label helps but does not replace such a notice.

Without the widget, the same video is available in `VoiceClient`: the `agentVideoChanged` event (`{ available }`) says whether video is there, and `attachAgentVideo(element)` puts it into your `<video>`.

Page requirements

- **HTTPS.** Without a secure connection the browser does not grant the microphone, and `startCall` fails with `WEBRTC_UNSUPPORTED`.
- **Content Security Policy.** If your site has a CSP, allow: the host the SDK files load from — in `script-src` and `style-src`; your server and the address from the session's `url` field (HTTPS and WSS) — in `connect-src`; `blob:` — in `media-src`. For the avatar: the placeholder image host — in `img-src`, the clip host — in `media-src`; with clips also `style-src-attr 'unsafe-inline'`.
- **Browsers.** WebRTC support is required — every modern desktop and mobile browser has it.

TECHNICAL REFERENCE

Call outcomes: `call_status` and `status_reason`

The outcome axis: buckets, the reasons inside them, and how they fold into a verdict.

`call_status` — final status bucket of a finished call (empty string until finalized). One axis, one source of truth; quality (`call_successful`) and technical reliability (`health`) are separate axes and never leak into the status:

Value	Meaning
`completed`	the call connected and finished (in any way)
`not_connected`	an outbound call never connected, or the caller of an inbound call hung up before the agent answered
`error`	the platform could not carry the call through: a connected call was dropped by a platform failure, the agent never joined, the callee answered but no audio ever came from them, or the outcome was never finalized (details in `health`)

`status_reason` — fine-grained reason, always belonging to exactly one bucket:

The **Analysis** column says whether post-call analysis runs for the call — the summary, the extracted fields and the success verdict. These are the defaults: the list of reasons that skip analysis is an agent setting.

Bucket	Value	Meaning	Analysis
`completed`	`agent_hangup`	Conversation ran to the end of the agent's flow.	yes
`completed`	`user_hangup`	Callee spoke but hung up before the flow finished.	yes
`completed`	`no_speech`	Call connected but no recognizable callee speech.	no
`completed`	`too_short`	Below the agent's configured minimum turns/duration. The threshold is off by default, so this reason does not occur.	no
`completed`	`voicemail_reached`	Answering machine detected (check `message_left`).	only when a message was left
`completed`	`voicemail_suppressed`	Answering machine detected, but the message-cap rule forbade leaving a message.	no
`completed`	`max_duration_reached`	Auto-ended at the agent's maximum call duration.	yes
`completed`	`silence_timeout`	Auto-ended after prolonged callee silence.	yes
`completed`	`transferred_cold` / `transferred_warm`	Call successfully handed off; the AI left the conversation.	yes
`completed`	`observer_incident`	Auto-ended by a compliance observer's end-call reaction.	yes
`completed`	`platform_shutdown`	The platform ended the call while stopping for an update.	yes
`completed`	`screener_declined`	A call screener answered, and the callee's assistant declined the call or hung up while holding the line. No person spoke. Only for agents with call screen handling on.	no
`completed`	`screener_timeout`	A call screener answered, and the talk with it ran past the hold timeout or the screener phase timeout. No person spoke. Only with call screen handling on.	no
`completed`	`ivr_gate`	A key-press menu (IVR) answered. The agent does not press keys and ends the call. Only with call screen handling on.	no
`not_connected`	`no_answer`	The subscriber's phone rang and nobody picked up within the ring window.	no
`not_connected`	`busy`	Destination busy.	no

Bucket	Value	Meaning	Analysis
`not_connected`	`user_declined`	Callee explicitly declined the call.	no
`not_connected`	`invalid_destination`	Nonexistent or invalid number.	no
`not_connected`	`trunk_auth_failed`	SIP trunk authentication failed.	no
`not_connected`	`provider_unavailable`	The call never reached the subscriber: telephony did not answer in time or refused, and there was no ringing.	no
`not_connected`	`sip_routing_error`	SIP routing failure.	no
`not_connected`	`spam_blocked`	Blocked as spam by the carrier.	no
`not_connected`	`dial_failed`	Generic dial failure (no finer detail available). The number is redialed after the campaign interval; in expert logic this is the **Dial failed** condition.	no
`not_connected`	`cancelled_by_function`	A pre-dial function of the agent decided not to place the call.	no
`not_connected`	`caller_abandoned`	Inbound call: the caller hung up before the agent answered. No conversation took place; `duration` is `null`. Before 14.10.2026 such calls came as `completed` / `no_speech`.	no
`not_connected`	`cancelled_by_operator`	The dial was stopped by an outside command before it connected — the cabinet button or `POST /v1/calls/{id}/end`. Neither the contact attempt nor the frequency cap is spent on it.	no
`error`	`internal_error`	Platform failure on a connected call — see `health` / `health_reasons`.	no
`error`	`agent_unavailable`	The voice agent never joined the call.	no
`error`	`no_media`	The callee answered, but no audio ever came from their side (the carrier sent no media for the whole call). `duration` is real; there is no transcript. Outbound calls only, and only when the agent has **No media detection** turned on (on by default); with it turned off, such a call comes as `no_speech` or `silence_timeout`, like a silent callee.	no
`error`	`not_finalized`	The outcome was never recorded; `duration` is `null`.	no

Which carrier answer gives which reason

The reasons of the `not_connected` bucket come from the carrier's answer to our INVITE. How fine-grained that is depends on what the carrier's server publishes: with no code, the reason stays the generic `dial_failed` and the code itself sits in `error_code` beside it.

Reason	SIP codes
`no_answer`	408, 480, 487
`busy`	486, 600
`user_declined`	603
`invalid_destination`	404, 410, 484, 604
`spam_blocked`	607, 608
`provider_unavailable`	488, 493, 500, 502, 503, 504
`trunk_auth_failed`	401; 403 on a registration trunk
`sip_routing_error`	380, 400, 483
`dial_failed`	everything else, including 403 on a peer-to-peer trunk

A 403 on a peer-to-peer (IP-based) trunk is a failed dial, not a trunk failure. Such a trunk has no credentials, so a 403 means the carrier does not let the call into that destination. The call gets `dial_failed`, and the 403 is in `error_code`. On

a registration trunk, and when the trunk is not found, a 403 stays `trunk_auth_failed`.

`606 Not Acceptable` is deliberately left in `dial_failed`: the same code arrives both from a carrier out of capacity and from an answering machine rejecting the call. It has no one honest reason, so read `error_code` for such calls.

How the reason is chosen

The rules apply top down, and the first one that fires wins:

1. **The call never connected** — a `not_connected` reason from the carrier's answer; for an inbound call where the caller left before the agent answered — `caller_abandoned`.
2. **The platform decided the outcome** — answering machine, call screener or key-press menu, handoff to an operator, silence, the duration limit, an observer incident or a stop for an update.
3. **Not a single recognized callee turn** — `no_speech`; fewer turns or seconds than the configured threshold — `too_short`.
4. **Otherwise it comes down to who hung up** — the flow reached its end (`agent_hangup`) or the callee hung up first (`user_hangup`).

> **`busy` and `user_declined` are distinct reasons.** `busy` means busy and `user_declined` means the subscriber actually declined the call. If you build reports on dial-failure reasons, separate the two values rather than folding them into one.
>
> A second consequence for retry logic on your side: **whether a reason is permanent depends on the market**. `user_declined` closes a number for good only on the US profile; on other profiles it is transient, and the number returns to dialing having spent an attempt. The campaign launch check refuses an `expert` rule that retries a permanent outcome (`rule_permanent_retry`).

What a campaign does with the number after a call

This is how standard campaign logic decides. In expert logic an outcome is handled the same way when no rule matches it or its rule has used up `max_matches`; an explicit rule — `stop_calling`, `retry_in` or `wait_for_pass_end` — decides instead.

What happened	`status_reason`	What happens to the number	Attempt spent
Not reached	`no_answer`, `busy`, `dial_failed`	redial after the campaign interval	yes
Answered, but not by a person	`voicemail_reached`, `voicemail_suppressed`, `screener_timeout`, `ivr_gate`	redial after the campaign interval	yes
The callee or their call screener declined	`user_declined`, `screener_declined`	under the US profile — number closed; under others — redial after the interval	yes
The number is unusable	`invalid_destination`, `spam_blocked`	number closed	yes
The platform failed on a connected call	`internal_error`, `agent_unavailable`, `no_media`, `not_finalized`	redial after the campaign interval	yes
Telephony failed before reaching the callee	`trunk_auth_failed`, `provider_unavailable`, `sip_routing_error`	by default, redial with a growing pause: 1, 2, 4, 8 minutes; after the fifth such failure the number is "Exhausted"	no
A conversation took place	`agent_hangup`, `user_hangup`, `no_speech`, `too_short`, `silence_timeout`, `max_duration_reached`, `transferred_cold`, `transferred_warm`, `observer_incident`, `platform_shutdown`	number closed	yes

What happened	`status_reason`	What happens to the number	Attempt spent
The call was cancelled before the dial	`cancelled_by_function`, `cancelled_by_operator`	number closed	no

- **A redial is scheduled only while dials are left and the campaign has not ended.** If `attempts` is used up or the next dial would fall after `ends_at`, the number becomes "Exhausted".
- **The interval** is the campaign interval or the step of the `intervals_s` ladder for this retry (Campaign fields). The calling window, the do-not-call registry and the frequency caps apply to every redial.
- **Voicemail is redialed whether or not a message was left.** How many messages to leave is decided by the voicemail message policy, not by this table.
- **An unknown reason closes the number:** if a new value appears in the axis, the campaign will not redial for a reason it does not understand.

What gets confused most often

`silence_timeout` **and** `no_speech` **are one situation with two endings.** The callee is silent in both; what differs is who hung up. The platform — `silence_timeout`; the callee first — `no_speech`. If you count empty calls, add the two together.

A callee who hangs up without a word is `no_speech`**, not** `user_hangup`**.** The second reason is used only where a conversation took place. An inbound caller who hangs up before the agent answers is `caller_abandoned`.

Unrecognized speech counts as silence. Noise, speech too quiet to recognize and a drop before a single recognized word all give the same reasons as silence.

`silence_timeout` **is analyzed,** `no_speech` **is not.** That is not a quirk of the setting: the callee may have talked and then gone quiet, so silence at the end does not mean an empty call.

The silence threshold is counted from the agent's last utterance, reminders included. Every reminder along the lines of "Are you still there?" restarts the count. With a 12-second threshold and two reminders 8 seconds apart, a call with a silent callee runs about 30 seconds, not 12. Once the reminders run out, the call ends one reminder interval after the last one — even when the silence threshold is longer.

A retry rule on the answering machine catches both cases. A condition on `voicemail_reached` also fires when the reason was `voicemail_suppressed`.

The three screener reasons are `completed`**, not** `not_connected`**.** The line was answered — by the callee's assistant or a menu — but no person spoke. When a screener connects a person, the call gets the usual reason of the conversation that followed; when it offers to leave a message and the agent leaves one, the reason is `voicemail_reached`.

`health_reasons` — codes explaining a `warning` / `error` health verdict. A call's verdict is the worst code that fired: any error-level code makes the call `error`, otherwise `warning`. The vocabulary is open — new codes are added over time, so treat an unfamiliar one as data, not as a parsing failure.

Codes come from four sources. The split matters: `health_reasons` **carries the output of a rule, never its input** — `status_reason` values and the internal codes of call events never appear here.

1. Call events — something went wrong inside the conversation:

Code	Level	When
`llm_execution_failed`	error	the execution model did not answer and the agent spoke its fallback line
`llm_role_failed`	warning	a model in a supporting role did not answer
`llm_retry`	warning	the answer came from the fallback model
`context_overflow`	error	the model's context overflowed

Code	Level	When
`tts_unspoken`	error	synthesis failed and the line was never spoken
`tts_error`	warning	a synthesis failure the call recovered from
`tts_fallback`	warning	switched to the reserve voice
`asr_turn_lost`	error	the caller's turn was lost by recognition
`asr_error`	warning	a recognition failure that lost no turn
`realtime_error`	error	a failure in realtime (speech-to-speech) mode
`noisy_line`	warning	the line was noisy: the call ran, recognition did not
`transfer_failed`	error	the transfer broke down. A caller who hangs up during the transfer announcement is not a failure: the transfer outcome is `caller_left`, with no mark
`transfer_ring_timeout`	warning	the destination did not answer and the fallback edge ran
`observer_error`	warning	the compliance observer failed
`normalizer_degraded`	warning	the pre-TTS normalizer did not answer
`lang_detect_degraded`	warning	language detection did not answer
`rag_degraded`	warning	a knowledge base lookup failed
`token_budget_degraded`	warning	the context was trimmed to the token budget
`tool_failed`	warning	a function or code node returned an error
`silence_timeout`	warning	auto-ended on caller silence
`max_duration`	warning	auto-ended on maximum call duration

2. The call status axis — `status_reason` values fold into these:

Code	Level	From which `status_reason`
`disconnect_error`	error	`internal_error`, `not_finalized`, `agent_unavailable`, `no_media`
`dial_infra_failed`	error	`trunk_auth_failed`, `provider_unavailable`, `sip_routing_error`
`dial_failed`	warning	every other reason in the `not_connected` bucket

Reasons that belong to the callee (`no_answer`, `busy`, `user_declined`, `invalid_destination`, `spam_blocked`) dials you cancelled yourself before dialing (`cancelled_by_function`, `cancelled_by_operator`) and inbound calls abandoned before the answer (`caller_abandoned`) contribute **nothing**: they are not platform defects.

3. Post-call analysis — `analysis_skipped`, `analysis_failed`, `analysis_interrupted`, all warning level, from the terminal state of the analysis.

4. Recording and timing — `recording_head_lost` (part of the recording does not exist) and `slow_turns` (the 90th percentile of turn duration is above the configured threshold), both warning.

TECHNICAL REFERENCE

Webhooks

The `call_ended` and `call_analyzed` events, signing, retries and delivery order.

Webhooks are the primary channel for results; polling `GET /v1/calls/{call_id}` is the fallback. An administrator registers the endpoint **on the agent**: open the agent in the editor → **Settings** tab → **Quality & integrations** card → section **Webhooks**; the form is described field by field on the Webhooks settings page. Payload examples are on the Webhooks page.

- **URL** — `http://` or `https://` (the platform does not enforce HTTPS; use it anyway). Delivery is subject to the platform's outbound host policy: private addresses and disallowed hosts may be refused by the operator's configuration, and such deliveries fail into the dead-letter queue.
- **Event subscription** — `call_ended`, `call_analyzed` or both (default: both).
- **Signing secret** — a **reference to a server-side secret**: in the form the administrator enters only the name of an environment variable of the platform deployment. The value itself is set on the platform side, never entered in the cabinet, and shared with you out of band.
- **Agent** — an endpoint is bound to exactly one live agent of the organization. The field is mandatory: an empty agent name is refused with `webhooks.agent_required`, an unknown or archived one with `webhooks.agent_not_found`. There is no organization-wide level — to cover several agents, register the endpoint on each of them (the same URL may be reused; `agent_id` in the payload tells them apart).
- **Test events** — an opt-in flag makes the endpoint also receive events of editor test sessions (`mode: "test"`).
- **Timeout** — how long the platform waits for your `2xx` (platform default 10 s, configurable per endpoint up to 120 s; `0` means the platform default).

Event envelope

Every event is an HTTP(S) `POST` with a JSON body (UTF-8, non-ASCII characters unescaped).

Field	Description
`event`	`call_ended` | `call_analyzed` (plus `ping` for test deliveries)
`event_id`	Unique event ID (`evt-` + 32 hex). Identical across delivery retries and across endpoints — **deduplicate on it**.
`timestamp`	When the event was built, epoch seconds.
`call_id`	Correlates with the call object.
`agent_id`	Agent name.
`flow_version`, `flow_version_id`	Flow version marker (same type caveat as in the call object) and published version ID.
`metadata`	Your payload from call creation — present in **every** event. For a campaign call the platform fills it itself, see below.

Campaign calls

A call made by a campaign carries in `metadata` the campaign and the contact it was dialed for. You don't need a separate `GET /v1/campaigns/{campaign_id}` to learn the campaign's name or where it was created.

Key	Description
`campaign_id`	The campaign.
`contact_id`	The campaign contact. Use it to address the contact in the contact operations.
`attempt_no`	The dial number for this contact, starting from 1. Redials after a technical failure count too.

Key	Description
`campaign_name`	The campaign name at the moment of dialing. Renaming the campaign later does not change it in calls already made.
`campaign_created_via`	Where the campaign was created: `api` — through the Web API, `web` — in the cabinet, `legacy` — through the compatibility adapter, `""` — not recorded (old campaigns).
`campaign_launched_via`	Where the campaign was launched — same values.

The keys are always strings, except `attempt_no`, which is a number. A call that is not part of a campaign does not carry them.

```
"metadata": {
  "campaign_id": "cam-b9b536a86e",
  "contact_id": "cc-41d7e0a2b9",
  "attempt_no": 2,
  "campaign_name": "October reminders",
  "campaign_created_via": "api",
  "campaign_launched_via": "web"
}
```

Headers on every delivery:

```
Content-Type: application/json
User-Agent: conversation-flow-webhooks/1.0
X-CFlow-Event: call_ended
X-CFlow-Event-Id: evt-4f0c22b17a9d4e3c8b6a5f019e2d7c31
X-CFlow-Timestamp: 1753344187
X-CFlow-Signature: v1=6f2a45c1e8…
```

Which event arrives when

- **`call_ended`** — as soon as the session finishes, before analysis and usually before the recording is finalized. This is the early notification: transcript, `call_status`, `status_reason`, `error_code`, and the timing fields (`answered_at`, `media_started_at`, `talk_from`, `talk_to`). `latency` is present only when per-turn metrics were collected; its keys are the raw metric names ending in `_ms` (`turn_ms`, `llm_ms`, `asr_ms`, `tts_ms`, …) with values `{avg, max}` in milliseconds.
- **`call_analyzed`** — when **both** post-call pipelines are done: the analysis (terminal in any of `done`, `skipped`, `failed`, `interrupted` — the event is always delivered) and the recording finalization. The event is **self-contained**: full call block + analysis + ready recording. If you only need the final outcome, subscribe to `call_analyzed` alone and skip `call_ended`. Its `call` block is identical to `call_ended.call` except that the recording stays top-level and is not duplicated inside `call`.
- **`ping`** — a test delivery from the cabinet button; `agent_id` is the agent the endpoint is bound to. Respond `2xx` and verify its signature like a real event. It is not retried.

When `analysis_status` is anything but `done`, `call_analysis` is `{}` and a top-level `reason` field explains why (for example `voicemail`, `no_speech`, `analytics_disabled`); `analytics_tier`, `analytics_model_used` and `analytics_language` are empty strings then.

Delivery semantics

- **At-least-once.** A delivery counts as successful only on a `2xx` from your endpoint; anything else (non-`2xx`, timeout, connection error) is retried. You **will** receive duplicates — deduplicate on `event_id`.
- **Retries with exponential backoff** — platform settings, not per endpoint: by default up to 8 attempts with pauses of 30, 60, 120, 240, 480, 960 and 1920 seconds. After the final failed attempt the event moves to a dead-letter queue, from which an administrator can redeliver it manually (History → session details → webhook deliveries). Dead deliveries are purged after a retention period, so the manual-retry window is finite.

- **Deliveries always go to the endpoint's current URL**, so a mistyped URL can be corrected and the event redelivered; an endpoint that was deleted or disabled sends its pending events straight to the dead-letter queue.
- **Ordering is best-effort.** `call_ended` is enqueued before `call_analyzed`, but delivery order is not guaranteed. Treat each event independently, correlating by `call_id`.
- **Respond fast.** Return `2xx` immediately after signature verification and process asynchronously: the platform waits at most the endpoint's timeout.

Signature verification

```
signed_payload = "{X-CFlow-Timestamp}." + raw request body
X-CFlow-Signature: v1=<hex( HMAC-SHA256(secret, signed_payload) )>
```

`X-CFlow-Timestamp` is an integer number of Unix seconds. Verify on the **raw bytes** of the body (before any JSON parsing or re-serialization — the body is UTF-8 with unescaped non-ASCII characters), compare in constant time, and reject stale timestamps — a 300-second window. Retried deliveries are re-signed with a fresh timestamp, so the window does not conflict with retries.

Python

```python
import hashlib
import hmac
import time

def verify_webhook(
    secret: str,
    signature_header: str,   # X-CFlow-Signature, e.g. "v1=6f2a45…"
    timestamp_header: str,   # X-CFlow-Timestamp, e.g. "1753344187"
    raw_body: bytes,         # exact request body bytes
    tolerance_seconds: int = 300,
) -> bool:
    try:
        ts = int(timestamp_header)
    except (TypeError, ValueError):
        return False
    if abs(time.time() - ts) > tolerance_seconds:
        return False  # replay protection
    signed = f"{ts}.".encode("utf-8") + raw_body
    expected = "v1=" + hmac.new(secret.encode("utf-8"), signed, hashlib.sha256).hexdigest()
    return hmac.compare_digest(expected, signature_header or "")
```

Node.js

```javascript
const crypto = require("crypto");

function verifyWebhook(secret, signatureHeader, timestampHeader, rawBody, toleranceSeconds = 300) {
  const ts = parseInt(timestampHeader, 10);
  if (!Number.isFinite(ts)) return false;
  if (Math.abs(Date.now() / 1000 - ts) > toleranceSeconds) return false;
  const expected =
    "v1=" +
    crypto
      .createHmac("sha256", secret)
      .update(`${ts}.`)
      .update(rawBody) // Buffer with the exact request bytes
      .digest("hex");
  const provided = Buffer.from(signatureHeader || "");
  const wanted = Buffer.from(expected);
  return provided.length === wanted.length && crypto.timingSafeEqual(provided, wanted);
}
```

In Python (Flask) use `request.get_data()`; in Node (Express) use `express.raw({ type: "application/json" })` — both preserve the exact bytes the signature was computed over.

Recordings

`recording_url` is a **stable** URL (`https://{{HUBTALK_FQDN}}/api/recordings/rec-…`): it does not expire and can be persisted in your systems. Each `GET` to it answers `307` with a fresh short-lived download link, so a URL saved months ago keeps working — until the recording itself is deleted by the retention policy (`recording.expires_at`). Other answers: `409` while the recording is not ready yet (keep polling), `410` after deletion, `503` when the storage is temporarily unavailable. Append `?download=1` for an attachment disposition.

The URL needs the API key of the organization the recording belongs to, in the same header as `/v1`: `Authorization: Bearer <API key>`. Without a key the answer is `401` `api_key_required`; an unknown or revoked key gets `401` `api_key_unknown` / `api_key_revoked`, and the `/v1` rate limit applies (`429`). A recording of another organization answers `404`, the same as a missing one. A browser that opens the URL without a key is sent to the cabinet: after sign-in it lands on the call card.

One edge case: if the recording takes unusually long to finalize (longer than the platform's wait, 30 s by default), `call_analyzed` is still emitted — with the current non-terminal recording status and an empty `recording_url`. The stable URL starts working on its own once the recording is ready.

TECHNICAL REFERENCE

Error codes and an end-to-end example

The full list of codes and a working scenario in one piece.

Error codes

By type

Status	`error.type`	Codes on the endpoints of this guide
400	`invalid_request_error`	`caller_number_not_allowed`, `caller_number_outside_grant_pool`, `destination_number_invalid`, `ring_duration_invalid`, `invalid_filter_value`, `invalid_cursor`
401	`authentication_error`	`api_key_required`, `api_key_unknown`, `api_key_revoked`
403	`permission_error`	`agent_not_in_subgroup`, `destination_not_allowed`, `privileged_key_required`, `root_org_key_required`
404	`not_found_error`	`trunk_not_found`, `subgroup_not_found`, `call_not_found`, `campaign_not_found`, `import_not_found`, `agent_not_found` (web calls only)
409	`conflict_error`	`agent_subgroup_ambiguous`, `agent_not_published`, `dialing_stopped`, `launch_blocked`, `campaign_transition_invalid`, `campaign_closed`, `market_package_not_installed`, `idempotency_key_in_flight`, `dnc_decision_without_upload`, `dnc_screen_required`
422	`invalid_request_error`	`invalid_request` (schema) and the campaign codes below
429	`rate_limit_error`	`api_key_rate_limit_exceeded`, `dial_queue_full`, `subgroup_concurrency_limit`, `subgroup_rate_limit`
500	`api_error`	`internal_error`
502	`api_error`	`dial_failed` (synchronous fallback only), `contact_file_storage_unavailable`
503	`service_unavailable_error`	`voice_backend_unavailable`, `agent_dispatch_failed` (web calls), `db_pool_exhausted` (the platform is briefly overloaded; retry after the `Retry-After` interval, 1 s)

Campaign error codes

Status	`error.code`	Cause
404	`campaign_not_found` / `import_not_found`	No such object **in your organization**; anything outside that boundary is indistinguishable from missing.
409	`launch_blocked`	Launch or resume checks did not pass; `error.blockers` names them.
409	`campaign_transition_invalid`	The transition is impossible from the current state.
409	`campaign_closed`	`calls`, `contacts/withdraw`, speed `PATCH`: the campaign is completed or cancelled.
409	`market_package_not_installed`	Contacts cannot be screened for this organization; contact support.
409	`idempotency_key_in_flight`	A request with this key is still being processed; wait for its answer.
409	`dnc_decision_without_upload`	A `dnc` decision on a campaign that has no import.
409	`dnc_screen_required`	The import has no source-row projection (pre-feature import): on a `dnc` decision, on `?only=dnc_match`, or as a launch blocker. Upload again.

Status	error.code	Cause
422	`invalid_request`	Body fails schema validation; `message` is an array of field issues.
422	`idempotency_key_required`	The `Idempotency-Key` header is missing or blank.
422	`idempotency_key_reuse`	This key was already used for a **different** request body.
422	`campaign_agent_unknown`	No agent with this name in your organization.
422	`campaign_window_invalid`	`ends_at` is not later than `starts_at`.
422	`campaign_timezone_unknown`	Not an IANA zone name.
422	`campaign_classification_invalid`	`classification` is not one of the three documented values.
422	`campaign_spec_invalid`	Name length, speed outside 1–200 (on creation and in `PATCH`), retry configuration, working-hours form or duplicate action rejected.
422	`campaign_open_ends_at_required` / `campaign_open_ends_at_too_far`	A campaign open to call additions has no `ends_at`, or one beyond the horizon.
422	`campaign_auto_speed_ends_at_required` / `campaign_auto_speed_working_hours_required`	A campaign with `auto_speed: true` has no `ends_at` or, when it runs over several days, no `working_hours`. On creation and on `PATCH`.
422	`dialing_route_unresolved` / `dialing_route_ambiguous`	No binding, or more than one candidate.
422	`dialing_binding_incomplete`	`route_id` or `from_number` named without `trunk_id`.
422	`working_hours_wider_than_law`	The schedule is wider than the floor (legal window or organization calling window).
422	`attestation_import_id_required`	`attestation.import_id` missing on `launch`.
422	`dnc_decision_invalid`	`dnc` is neither `block` nor `skip`.
422	`contact_rows_rejected`	The planned calls could not be stored at all; read the import report.
422	`report_format_invalid`	`format` is neither `csv` nor `xlsx`.
422	`report_filter_invalid`	`only` is neither omitted nor `dnc_match`.
422	`calls_required` / `calls_batch_too_large`	`calls` is empty, or has more rows than the cap (1000 by default).
422	`attestation_invalid`	`calls`: `attestation` is not exactly `{"confirmed": true}`.
422	`calls_rejected`	The batch was refused whole because of a data error (or a structural parse refusal); `error.import` names the rows.
422	`calls_dnc_matched`	The batch was refused whole because at least one number is on the do-not-call registry; read the `dnc_match` report.
502	`contact_file_storage_unavailable`	The contact source could not be stored, so the upload was not accepted; retry.

End-to-end example

```
BASE_URL="https://app.hubtalk.ai"   # the host your cabinet runs on
API_KEY="cfk_…"   # the secret shown once when the key was created

# 1. Look up the agent's available lines (optional: omit the binding
#    entirely and the platform resolves it itself)
curl -s "$BASE_URL/v1/dialing-routes?agent_id=collections-agent" \
```

```
  -H "Authorization: Bearer $API_KEY"

# 2. Create the campaign, hand over the numbers and launch — one request
curl -s -X POST "$BASE_URL/v1/campaigns" \
  -H "Authorization: Bearer $API_KEY" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: june-collections-2026-09-09" \
  -d '{
        "name": "june collections",
        "agent_id": "collections-agent",
        "calls_per_minute": 10,
        "timezone": "Asia/Almaty",
        "classification": "debt_collection",
        "working_hours": [{"days": [0,1,2,3,4], "start": "09:00", "end": "19:00"}],
        "attestation": {"confirmed": true},
        "planned_calls": [
          {"phone": "+77001234567", "name": "Aigerim", "variables": {"debt": "15000"}},
          {"phone": "+77007654321", "name": "Daniyar"}
        ]
      }'
# → 201. Check "status": "running" | "scheduled" | "draft".
#   "draft" means the reason is named in "blockers": fix it and call
#   POST /v1/campaigns/{id}/launch with
#   {"attestation": {"confirmed": true, "import_id": "<import.import_id>"}}
#   Note: by default retry_config.attempts = 1 — nobody is called back.

# 3. Stream more numbers later (the campaign must not be closed)
curl -s -X POST "$BASE_URL/v1/campaigns/cam-b9b536a86e/calls" \
  -H "Authorization: Bearer $API_KEY" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: crm-batch-2026-09-09-17" \
  -d '{"calls": [{"phone": "+77009998877", "name": "Symbat"}],
       "attestation": {"confirmed": true}}'
# → 201 (accepted whole) or 422 calls_rejected / calls_dnc_matched (refused whole)

# 4. Follow it
curl -s "$BASE_URL/v1/campaigns/cam-b9b536a86e/progress" \
  -H "Authorization: Bearer $API_KEY"

# 5. Inspect the problematic rows
curl -s "$BASE_URL/v1/imports/imp-353d158dd2/report?format=csv" \
  -H "Authorization: Bearer $API_KEY" -o import-report.csv

# ...and the do-not-call matches on their own
curl -s "$BASE_URL/v1/imports/imp-353d158dd2/report?only=dnc_match&format=csv" \
  -H "Authorization: Bearer $API_KEY" -o import-dnc.csv

# 6. Per-call results arrive as the usual call_ended / call_analyzed webhooks —
#    a campaign dials the same calls this API places one by one
```