



≡ Messaging platforms > **Microsoft Teams**

Messaging platforms

Microsoft Teams

“Abandon all hope, ye who enter here.”

Updated: 2026-01-21

Status: text + DM attachments are supported; channel/group file sending requires `sharePointSiteId` + Graph permissions (see [Sending files in group chats](#)). Polls are sent via Adaptive Cards.

Plugin required

Microsoft Teams ships as a plugin and is not bundled with the core install.

Breaking change (2026.1.15): MS Teams moved out of core. If you use it, you must install the plugin.

Explainable: keeps core installs lighter and lets MS Teams dependencies update independently.

Install via CLI (npm registry):

```
openclaw plugins install @openclaw/msteams
```

Local checkout (when running from a git repo):

 `openclaw plugins install ./extensions/msteams`

If you choose Teams during configure/onboarding and a git checkout is detected, OpenClaw will offer the local install path automatically.

Details:

Quick setup (beginner)

1. Install the Microsoft Teams plugin.
2. Create an **Azure Bot** (App ID + client secret + tenant ID).
3. Configure OpenClaw with those credentials.
4. Expose `/api/messages` (port 3978 by default) via a public URL or tunnel.
5. Install the Teams app package and start the gateway.

Minimal config:

```
{
  channels: {
    msteams: {
      enabled: true,
      appId: "<APP_ID>",
      appPassword: "<APP_PASSWORD>",
      tenantId: "<TENANT_ID>",
      webhook: { port: 3978, path: "/api/messages" },
    },
  },
}
```

Note: group chats are blocked by default (`channels.msteams.groupPolicy: "allowlist"`). To allow group replies, set `channels.msteams.groupAllowFrom` (or use `groupPolicy: "open"` to allow any member, mention-gated).

Goals



Talk to OpenClaw via Teams DMs, group chats, or channels.

Keep routing deterministic: replies always go back to the channel they arrived on.

Default to safe channel behavior (mentions required unless configured otherwise).

Config writes

By default, Microsoft Teams is allowed to write config updates triggered by `/config set|unset` (requires `commands.config: true`).

Disable with:

```
{  
  channels: { msteams: { configWrites: false } },  
}
```

Access control (DMs + groups)


DM access

Default: `channels.msteams.dmPolicy = "pairing"` . Unknown senders are ignored until approved.

`channels.msteams.allowFrom` accepts AAD object IDs, UPNs, or display names. The wizard resolves names to IDs via Microsoft Graph when credentials allow.

Group access

Default: `channels.msteams.groupPolicy = "allowlist"` (blocked unless you add `groupAllowFrom`). Use `channels.defaults.groupPolicy` to override the default when unset.

 `channels.msteams.groupAllowFrom` controls which senders can trigger in group chats/channels (falls back to `channels.msteams.allowFrom`).

Set `groupPolicy: "open"` to allow any member (still mention-gated by default).

To allow **no channels**, set `channels.msteams.groupPolicy: "disabled"`.

Example:

```
{
  channels: {
    msteams: {
      groupPolicy: "allowlist",
      groupAllowFrom: ["user@org.com"],
    },
  },
}
```

Teams + channel allowlist

Scope group/channel replies by listing teams and channels under `channels.msteams.teams`.

Keys can be team IDs or names; channel keys can be conversation IDs or names.

When `groupPolicy="allowlist"` and a teams allowlist is present, only listed teams/channels are accepted (mention-gated).

The configure wizard accepts `Team/Channel` entries and stores them for you.

On startup, OpenClaw resolves team/channel and user allowlist names to IDs (when Graph permissions allow) and logs the mapping; unresolved entries are kept as typed.

Example:



```
channels: {  
  msteams: {  
    groupPolicy: "allowlist",  
    teams: {  
      "My Team": {  
        channels: {  
          General: { requireMention: true },  
        },  
      },  
    },  
  },  
}
```

How it works

1. Install the Microsoft Teams plugin.
2. Create an **Azure Bot** (App ID + secret + tenant ID).
3. Build a **Teams app package** that references the bot and includes the RSC permissions below.
4. Upload/install the Teams app into a team (or personal scope for DMs).
5. Configure `msteams` in `~/.openclaw/openclaw.json` (or env vars) and start the gateway.
6. The gateway listens for Bot Framework webhook traffic on `/api/messages` by default.

Azure Bot Setup (Prerequisites)

Before configuring OpenClaw, you need to create an Azure Bot resource.

Step 1: Create Azure Bot



1. Go to Create Azure Bot
2. Fill in the **Basics** tab:

Field	Value
Bot handle	Your bot name, e.g., <code>openclaw-msteams</code> (must be unique)
Subscription	Select your Azure subscription
Resource group	Create new or use existing
Pricing tier	Free for dev/testing
Type of App	Single Tenant (recommended - see note below)
Creation type	Create new Microsoft App ID

Deprecation notice: Creation of new multi-tenant bots was deprecated after 2025-07-31. Use **Single Tenant** for new bots.

3. Click **Review + create** → **Create** (wait ~1-2 minutes)

Step 2: Get Credentials

1. Go to your Azure Bot resource → **Configuration**
2. Copy **Microsoft App ID** → this is your `appId`
3. Click **Manage Password** → go to the App Registration
4. Under **Certificates & secrets** → **New client secret** → copy the **Value** → this is your `appPassword`
5. Go to **Overview** → copy **Directory (tenant) ID** → this is your `tenantId`

Step 3: Configure Messaging Endpoint

1. In Azure Bot → **Configuration**

2. Set **Messaging endpoint** to your webhook URL:



Production: `https://your-domain.com/api/messages`

Local dev: Use a tunnel (see [Local Development](#) below)

Step 4: Enable Teams Channel

1. In Azure Bot → **Channels**
2. Click **Microsoft Teams** → Configure → Save
3. Accept the Terms of Service

Local Development (Tunneling)

Teams can't reach `localhost` . Use a tunnel for local development:

Option A: ngrok

```
ngrok http 3978
# Copy the https URL, e.g., https://abc123.ngrok.io
# Set messaging endpoint to: https://abc123.ngrok.io/api/messages
```

Option B: Tailscale Funnel

```
tailscale funnel 3978
# Use your Tailscale funnel URL as the messaging endpoint
```

Teams Developer Portal (Alternative)

Instead of manually creating a manifest ZIP, you can use the

:

1. Click + **New app**
2. Fill in basic info (name, description, developer info)

3. Go to **App features** → **Bot**
4. Select **Enter a bot ID manually** and paste your Azure Bot App ID
5. Check scopes: **Personal**, **Team**, **Group Chat**
6. Click **Distribute** → **Download app package**
7. In Teams: **Apps** → **Manage your apps** → **Upload a custom app** → select the ZIP

This is often easier than hand-editing JSON manifests.

Testing the Bot

Option A: Azure Web Chat (verify webhook first)

1. In Azure Portal → your Azure Bot resource → **Test in Web Chat**
2. Send a message - you should see a response
3. This confirms your webhook endpoint works before Teams setup

Option B: Teams (after app installation)

1. Install the Teams app (sideload or org catalog)
2. Find the bot in Teams and send a DM
3. Check gateway logs for incoming activity

Setup (minimal text-only)

1. Install the Microsoft Teams plugin

From npm: `openclaw plugins install @openclaw/msteams`

From a local checkout: `openclaw plugins install ./extensions/msteams`

2. Bot registration

Create an Azure Bot (see above) and note:

App ID



Client secret (App password)

Tenant ID (single-tenant)

3. Teams app manifest

Include a `bot` entry with `botId = <App ID>` .

Scopes: `personal` , `team` , `groupChat` .

`supportsFiles: true` (required for personal scope file handling).

Add RSC permissions (below).

Create icons: `outline.png` (32x32) and `color.png` (192x192).

Zip all three files together: `manifest.json` , `outline.png` ,
`color.png` .

4. Configure OpenClaw

```
{  
  "msteams": {  
    "enabled": true,  
    "appId": "<APP_ID>",  
    "appPassword": "<APP_PASSWORD>",  
    "tenantId": "<TENANT_ID>",  
    "webhook": { "port": 3978, "path": "/api/messages" }  
  }  
}
```

You can also use environment variables instead of config keys:

`MSTEAMS_APP_ID`

`MSTEAMS_APP_PASSWORD`

`MSTEAMS_TENANT_ID`

5. Bot endpoint

Set the Azure Bot Messaging Endpoint to:

`https://<host>:3978/api/messages` (or your chosen path/port).

6. Run the gateway

The Teams channel starts automatically when the plugin is installed and `msteams` config exists with credentials.

History context

`channels.msteams.historyLimit` controls how many recent channel/group messages are wrapped into the prompt.

Falls back to `messages.groupChat.historyLimit`. Set `0` to disable (default 50).

DM history can be limited with `channels.msteams.dmHistoryLimit` (user turns). Per-user overrides: `channels.msteams.dms["<user_id>"].historyLimit`.

Current Teams RSC Permissions (Manifest)

These are the **existing resourceSpecific permissions** in our Teams app manifest. They only apply inside the team/chat where the app is installed.

For channels (team scope):

`ChannelMessage.Read.Group` (Application) - receive all channel messages without @mention

`ChannelMessage.Send.Group` (Application)

`Member.Read.Group` (Application)


`Owner.Read.Group` (Application)

`ChannelSettings.Read.Group` (Application)

`TeamMember.Read.Group` (Application)

`TeamSettings.Read.Group` (Application)

For group chats:

 ChatMessage.Read.Chat (Application) - receive all group chat messages without @mention

>

Example Teams Manifest (redacted)

Minimal, valid example with the required fields. Replace IDs and URLs.



```

"$schema": "https://developer.microsoft.com/en-us/json-schemas/teams/v1.23/Micro
"manifestVersion": "1.23",
"version": "1.0.0", >
"id": "00000000-0000-0000-0000-000000000000",
"name": { "short": "OpenClaw" },
"developer": {
  "name": "Your Org",
  "websiteUrl": "https://example.com",
  "privacyUrl": "https://example.com/privacy",
  "termsOfUseUrl": "https://example.com/terms"
},
"description": { "short": "OpenClaw in Teams", "full": "OpenClaw in Teams" },
"icons": { "outline": "outline.png", "color": "color.png" },
"accentColor": "#5B6DEF",
"bots": [
  {
    "botId": "11111111-1111-1111-1111-111111111111",
    "scopes": ["personal", "team", "groupChat"],
    "isNotificationOnly": false,
    "supportsCalling": false,
    "supportsVideo": false,
    "supportsFiles": true
  }
],
"webApplicationInfo": {
  "id": "11111111-1111-1111-1111-111111111111"
},
"authorization": {
  "permissions": {
    "resourceSpecific": [
      { "name": "ChannelMessage.Read.Group", "type": "Application" },
      { "name": "ChannelMessage.Send.Group", "type": "Application" },
      { "name": "Member.Read.Group", "type": "Application" },
      { "name": "Owner.Read.Group", "type": "Application" },
      { "name": "ChannelSettings.Read.Group", "type": "Application" },
      { "name": "TeamMember.Read.Group", "type": "Application" },
      { "name": "TeamSettings.Read.Group", "type": "Application" },
      { "name": "ChatMessage.Read.Chat", "type": "Application" }
    ]
  }
}

```



Manifest caveats (must-have fields)

`bots[].botId` **must** match the Azure Bot App ID.

`webApplicationInfo.id` **must** match the Azure Bot App ID.

`bots[].scopes` must include the surfaces you plan to use (`personal` , `team` , `groupChat`).

`bots[].supportsFiles: true` is required for file handling in `personal` scope.

`authorization.permissions.resourceSpecific` must include `channel read/send` if you want channel traffic.

Updating an existing app


To update an already-installed Teams app (e.g., to add RSC permissions):

1. Update your `manifest.json` with the new settings
2. **Increment the version field** (e.g., `1.0.0` → `1.1.0`)
3. **Re-zip** the manifest with icons (`manifest.json` , `outline.png` , `color.png`)
4. Upload the new zip:

Option A (Teams Admin Center): Teams Admin Center → Teams apps → Manage apps → find your app → Upload new version

Option B (Sideload): In Teams → Apps → Manage your apps → Upload a custom app

5. **For team channels:** Reinstall the app in each team for new permissions to take effect

6.  Fully quit and relaunch Teams (not just close the window) to clear cached app metadata

>

Capabilities: RSC only vs Graph

With Teams RSC only (app installed, no Graph API permissions)

Works:

- Read channel message **text** content.
- Send channel message **text** content.
- Receive **personal (DM)** file attachments.

Does NOT work:

- Channel/group **image or file contents** (payload only includes HTML stub).
- Downloading attachments stored in SharePoint/OneDrive.
- Reading message history (beyond the live webhook event).

With Teams RSC + Microsoft Graph Application permissions

Adds:

- Downloading hosted contents (images pasted into messages).
- Downloading file attachments stored in SharePoint/OneDrive.
- Reading channel/chat message history via Graph.

RSC vs Graph API

Capability	RSC Permissions	Graph API
Real-time messages	Yes (via webhook)	No (polling only)

Capability	RSC Permissions	Graph API
 Historical messages	No	Yes (can query history)
Setup complexity	App manifest only	Requires admin consent + token flow
Works offline	No (must be running)	Yes (query anytime)

Bottom line: RSC is for real-time listening; Graph API is for historical access. For catching up on missed messages while offline, you need Graph API with `ChannelMessage.Read.All` (requires admin consent).

Graph-enabled media + history (required for channels)

If you need images/files in **channels** or want to fetch **message history**, you must enable Microsoft Graph permissions and grant admin consent.

1. In Entra ID (Azure AD) **App Registration**, add Microsoft Graph **Application permissions**:

`ChannelMessage.Read.All` (channel attachments + history)

`Chat.Read.All` or `ChatMessage.Read.All` (group chats)

2. **Grant admin consent** for the tenant.
3. Bump the Teams app **manifest version**, re-upload, and **reinstall the app in Teams**.
4. **Fully quit and relaunch Teams** to clear cached app metadata.

Additional permission for user mentions: User @mentions work out of the box for users in the conversation. However, if you want to dynamically search and mention users who are **not in the current conversation**, add `User.Read.All` (Application) permission and grant admin consent.

Known Limitations

Webhook timeouts



Teams delivers messages via HTTP webhook. If processing takes too long (e.g., slow LLM responses), you may see:

Gateway timeouts

Teams retrying the message (causing duplicates)

Dropped replies

OpenClaw handles this by returning quickly and sending replies proactively, but very slow responses may still cause issues.

Formatting

Teams markdown is more limited than Slack or Discord:

Basic formatting works: **bold**, *italic*, `code` , links

Complex markdown (tables, nested lists) may not render correctly

Adaptive Cards are supported for polls and arbitrary card sends (see below)

Configuration

Key settings (see `/gateway/configuration` for shared channel patterns):

`channels.msteams.enabled` : enable/disable the channel.

`channels.msteams.appId` , `channels.msteams.appPassword` ,

`channels.msteams.tenantId` : bot credentials.

`channels.msteams.webhook.port` (default `3978`)

`channels.msteams.webhook.path` (default `/api/messages`)

`channels.msteams.dmPolicy` : `pairing` | `allowlist` | `open` | `disabled`
(default: `pairing`)



`channels.msteams.allowFrom` : allowlist for DMs (AAD object IDs, UPNs, or display names). The wizard resolves names to IDs during setup when Graph access is available.

`channels.msteams.textChunkLimit` : outbound text chunk size.

`channels.msteams.chunkMode` : `length` (default) or `newline` to split on blank lines (paragraph boundaries) before length chunking.

`channels.msteams.mediaAllowHosts` : allowlist for inbound attachment hosts (defaults to Microsoft/Teams domains).

`channels.msteams.mediaAuthAllowHosts` : allowlist for attaching Authorization headers on media retries (defaults to Graph + Bot Framework hosts).

`channels.msteams.requireMention` : require @mention in channels/groups (default true).

`channels.msteams.replyStyle` : `thread` | `top-level` (see [Reply Style](#)).

`channels.msteams.teams.<teamId>.replyStyle` : per-team override.

`channels.msteams.teams.<teamId>.requireMention` : per-team override.

`channels.msteams.teams.<teamId>.tools` : default per-team tool policy overrides (`allow` / `deny` / `alsoAllow`) used when a channel override is missing.

`channels.msteams.teams.<teamId>.toolsBySender` : default per-team per-sender tool policy overrides (`"*"` wildcard supported).

`channels.msteams.teams.<teamId>.channels.<conversationId>.replyStyle` : per-channel override.

`channels.msteams.teams.<teamId>.channels.<conversationId>.requireMention` : per-channel override.

`channels.msteams.teams.<teamId>.channels.<conversationId>.tools` : per-channel tool policy overrides (`allow` / `deny` / `alsoAllow`).

`channels.msteams.teams.<teamId>.channels.<conversationId>.toolsBySender` : per-channel per-sender tool policy overrides (`"*"` wildcard supported).



`channels.msteams.sharePointSiteId` : SharePoint site ID for file uploads in group chats/channels (see [Sending files in group chats](#)).

>

Routing & Sessions

Session keys follow the standard agent format (see [/concepts/session](#)):

Direct messages share the main session (`agent:<agentId>:<mainKey>`).

Channel/group messages use conversation id:

`agent:<agentId>:msteams:channel:<conversationId>`

`agent:<agentId>:msteams:group:<conversationId>`

Reply Style: Threads vs Posts

Teams recently introduced two channel UI styles over the same underlying data model:

Style	Description	Recommended replyStyle
Posts (classic)	Messages appear as cards with threaded replies underneath	<code>thread</code> (default)
Threads (Slack-like)	Messages flow linearly, more like Slack	<code>top-level</code>

The problem: The Teams API does not expose which UI style a channel uses. If you use the wrong `replyStyle` :

`thread` in a Threads-style channel → replies appear nested awkwardly

`top-level` in a Posts-style channel → replies appear as separate top-level posts instead of in-thread

Solution: Configure `replyStyle` per-channel based on how the channel is set up:

```
{
  >
  "msteams": {
    "replyStyle": "thread",
    "teams": {
      "19:abc...@thread.tacv2": {
        "channels": {
          "19:xyz...@thread.tacv2": {
            "replyStyle": "top-level"
          }
        }
      }
    }
  }
}
```

Attachments & Images

Current limitations:

DMs: Images and file attachments work via Teams bot file APIs.

Channels/groups: Attachments live in M365 storage (SharePoint/OneDrive). The webhook payload only includes an HTML stub, not the actual file bytes. **Graph API permissions are required** to download channel attachments.

Without Graph permissions, channel messages with images will be received as text-only (the image content is not accessible to the bot). By default, OpenClaw only downloads media from Microsoft/Teams hostnames. Override with `channels.msteams.mediaAllowHosts` (use `["*"]` to allow any host). Authorization headers are only attached for hosts in `channels.msteams.mediaAuthAllowHosts` (defaults to Graph + Bot Framework hosts). Keep this list strict (avoid multi-tenant suffixes).

Sending files in group chats



Bots can send files in DMs using the `FileConsentCard` flow (built-in). However, **sending files in group chats/channels** requires additional setup:

Context	How files are sent	Setup needed
DMs	<code>FileConsentCard</code> → user accepts → bot uploads	Works out of the box
Group chats/channels	Upload to SharePoint → share link	Requires <code>sharePointSiteId</code> + Graph permissions
Images (any context)	Base64-encoded inline	Works out of the box

Why group chats need SharePoint

Bots don't have a personal OneDrive drive (the `/me/drive` Graph API endpoint doesn't work for application identities). To send files in group chats/channels, the bot uploads to a **SharePoint site** and creates a sharing link.

Setup

1. **Add Graph API permissions** in Entra ID (Azure AD) → App

Registration:

`Sites.ReadWrite.All` (Application) - upload files to SharePoint

`Chat.Read.All` (Application) - optional, enables per-user sharing links

2. **Grant admin consent** for the tenant.

3. **Get your SharePoint site ID:**



```
# Via Graph Explorer or curl with a valid token:
curl -H "Authorization: Bearer $TOKEN" \
    "https://graph.microsoft.com/v1.0/sites/{hostname}/{site-path}"
>

# Example: for a site at "contoso.sharepoint.com/sites/BotFiles"
curl -H "Authorization: Bearer $TOKEN" \
    "https://graph.microsoft.com/v1.0/sites/contoso.sharepoint.com:/sites/BotFi

# Response includes: "id": "contoso.sharepoint.com,guid1,guid2"
```

4. Configure OpenClaw:

```
{
  channels: {
    msteams: {
      // ... other config ...
      sharePointSiteId: "contoso.sharepoint.com,guid1,guid2",
    },
  },
}
```

Sharing behavior

Permission	Sharing behavior
Sites.ReadWrite.All only	Organization-wide sharing link (anyone in org can access)
Sites.ReadWrite.All + Chat.Read.All	Per-user sharing link (only chat members can access)

Per-user sharing is more secure as only the chat participants can access the file. If Chat.Read.All permission is missing, the bot falls back to organization-wide sharing.

Fallback behavior



Scenario	Result
Group chat + file + <code>sharePointSiteId</code> configured	Upload to SharePoint, send sharing link
Group chat + file + no <code>sharePointSiteId</code>	Attempt OneDrive upload (may fail), send text only
Personal chat + file	FileConsentCard flow (works without SharePoint)
Any context + image	Base64-encoded inline (works without SharePoint)

Files stored location

Uploaded files are stored in a `/OpenClawShared/` folder in the configured SharePoint site's default document library.

Polls (Adaptive Cards)

OpenClaw sends Teams polls as Adaptive Cards (there is no native Teams poll API).

CLI: `openclaw message poll --channel msteams --target conversation:<id> ...`

Votes are recorded by the gateway in `~/.openclaw/msteams-polls.json`.

The gateway must stay online to record votes.

Polls do not auto-post result summaries yet (inspect the store file if needed).

Adaptive Cards (arbitrary)

Send any Adaptive Card JSON to Teams users or conversations using the `message` tool or CLI.

The `card` parameter accepts an Adaptive Card JSON object. When `card` is provided, the message text is optional.

Agent tool: >

```
{
  "action": "send",
  "channel": "msteams",
  "target": "user:<id>",
  "card": {
    "type": "AdaptiveCard",
    "version": "1.5",
    "body": [{ "type": "TextBlock", "text": "Hello!" }]
  }
}
```

CLI:

```
openclaw message send --channel msteams \
  --target "conversation:19:abc...@thread.tacv2" \
  --card '{"type":"AdaptiveCard","version":"1.5","body":[{"type":"TextBlock","text": "Hello!"}]}'
```

See [Adaptive Card schema](#) for card schema and examples. For target format details, see [Adaptive Card target format](#) below.

Target formats

MSTeams targets use prefixes to distinguish between users and conversations:

Target type	Format	Example
User (by ID)	user:<aad-object-id>	user:40a1a0ed-4ff2-4164-a219-55518990c197
User (by name)	user:<display-name>	user:John Smith (requires Graph API)

Target type	Format	Example
 Group/channel	conversation: <conversation-id>	conversation:19:abc123...@thread.tacv2
Group/channel (raw)	> <conversation-id>	19:abc123...@thread.tacv2 (if contains @thread)

CLI examples:

```
# Send to a user by ID
openclaw message send --channel msteams --target "user:40a1a0ed-..." --message "Hello!"

# Send to a user by display name (triggers Graph API lookup)
openclaw message send --channel msteams --target "user:John Smith" --message "Hello!"

# Send to a group chat or channel
openclaw message send --channel msteams --target "conversation:19:abc...@thread.tacv2" --message "Hello!"

# Send an Adaptive Card to a conversation
openclaw message send --channel msteams --target "conversation:19:abc...@thread.tacv2" --card '{"type":"AdaptiveCard","version":"1.5","body":[{"type":"TextBlock","text":"Hello!"}]}'
```

Agent tool examples:

```
{
  "action": "send",
  "channel": "msteams",
  "target": "user:John Smith",
  "message": "Hello!"
}
```



```
{  
  "action": "send",  
  "channel": "msteams",  
  "target": "conversation:19:abc...@thread.tacv2",  
  "card": {  
    "type": "AdaptiveCard",  
    "version": "1.5",  
    "body": [{ "type": "TextBlock", "text": "Hello" }]  
  }  
}
```

Note: Without the `user:` prefix, names default to group/team resolution. Always use `user:` when targeting people by display name.

Proactive messaging

Proactive messages are only possible **after** a user has interacted, because we store conversation references at that point.

See `/gateway/configuration` for `dmPolicy` and `allowlist gating`.

Team and Channel IDs (Common Gotcha)

The `groupId` query parameter in Teams URLs is **NOT** the team ID used for configuration. Extract IDs from the URL path instead:

Team URL:

```
https://teams.microsoft.com/l/team/19%3ABk4j...%40thread.tacv2/convers
```

Team ID (URL-decode this)

Channel URL:



https://teams.microsoft.com/l/channel/19%3A15bc...%40thread.tacv2/Char

Channel ID (URL-decode this)

For config:

Team ID = path segment after `/team/` (URL-decoded, e.g.,
`19:Bk4j...@thread.tacv2`)

Channel ID = path segment after `/channel/` (URL-decoded)

Ignore the `groupId` query parameter

Private Channels

Bots have limited support in private channels:

Feature	Standard Channels	Private Channels
Bot installation	Yes	Limited
Real-time messages (webhook)	Yes	May not work
RSC permissions	Yes	May behave differently
@mentions	Yes	If bot is accessible
Graph API history	Yes	Yes (with permissions)

Workarounds if private channels don't work:

1. Use standard channels for bot interactions
2. Use DMs - users can always message the bot directly
3. Use Graph API for historical access (requires
`ChannelMessage.Read.All`)

Troubleshooting



Common issues

Images not showing^g in channels: Graph permissions or admin consent missing. Reinstall the Teams app and fully quit/reopen Teams.

No responses in channel: mentions are required by default; set `channels.msteams.requireMention=false` or configure per team/channel.

Version mismatch (Teams still shows old manifest): remove + re-add the app and fully quit Teams to refresh.

401 Unauthorized from webhook: Expected when testing manually without Azure JWT - means endpoint is reachable but auth failed. Use Azure Web Chat to test properly.

Manifest upload errors

“Icon file cannot be empty”: The manifest references icon files that are 0 bytes. Create valid PNG icons (32x32 for `outline.png` , 192x192 for `color.png`).

“webApplicationInfo.Id already in use”: The app is still installed in another team/chat. Find and uninstall it first, or wait 5-10 minutes for propagation.

“Something went wrong” on upload: Upload via <https://admin.teams.microsoft.com> instead, open browser DevTools (F12) → Network tab, and check the response body for the actual error.

Sideload failing: Try “Upload an app to your org’s app catalog” instead of “Upload a custom app” - this often bypasses sideload restrictions.

RSC permissions not working

1. Verify `webApplicationInfo.id` matches your bot’s App ID exactly

2. Re-upload the app and reinstall in the team/chat
 3. Check if your org admin has blocked RSC permissions
-
4. Confirm you're using the right scope: `ChannelMessage.Read.Group` for teams, `ChatMessage.Read.Chat` for group chats
-

References

[Create Azure Bot](#) - Azure Bot setup guide

[Teams Developer Portal](#) - create/manage Teams apps

[Teams app manifest schema](#)

[Receive channel messages with RSC](#)

[RSC permissions reference](#)

[Teams bot file handling](#) (channel/group requires Graph)

[Proactive messaging](#)

< iMessage

LINE >

Powered by [mintlify](#)