Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

API2Channel: Automated C2 Feasibility Assessment of Cloud-Based and Public Legitimate Services

Chen Jun
Huazhong University of Science and Technology
Hongqi Chen
Huazhong University of Science and Technology
Weifeng Yuan
Huazhong University of Science and Technology
Feng Dong
dongfeng@hust.edu.cn
Huazhong University of Science and Technology
Haoyu Wang
Huazhong University of Science and Technology

Research Article

Keywords:

Posted Date: November 27th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11710567/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

API2Channel: Automated C2 Feasibility Assessment of Cloud-Based and Public Legitimate Services

Jun Chen [1], Hongqi Chen [1], Weifeng Yuan [1], Feng Dong [1*], Haoyu Wang [1]

[1]Huazhong University of Science and Technology, Wuhan, China.

*Corresponding author(s). E-mail(s): dongfeng@hust.edu.cn;
Contributing authors: jun_c@hust.edu.cn; hongqichen@hust.edu.cn; cougarsecu@gmail.com; haoyuwang@hust.edu.cn;

Abstract

Public APIs of **cloud-based and public legitimate services (CPLS)** can be abused to establish covert command-and-control (C2) channels. Existing studies mainly construct C2 implementations for selected services or analyze observed abuse, but lack an automated method for assessing communication feasibility before abuse is observed. The central challenge is that individual API functions, such as reading and writing data, may not compose into a complete bidirectional channel. We present **API2Channel**, a system that assesses C2 communication feasibility from public API specifications and official documentation. It analyzes resource relationships and API operations to determine whether a controller and an agent can exchange tasks and results. For services predicted feasible, it generates communication code and tests it through simulated API interactions and execution-guided repair. We evaluate 24 CPLS. API2Channel achieves 91.67% accuracy, compared with 66.67% and 83.33% for two large language model (LLM) baselines. API2Channel generated implementations for all 14 services it classified as feasible. In real-service tests, 12 implementations passed the core functional tests and platform controls blocked the other two. These results show how structured API analysis, automated construction, and execution-based validation can assess potential C2 abuse and test the resulting predictions on real services.

Keywords: Command and control, CPLS, API analysis, Large language models, Automated software construction, Contract-based testing

1 Introduction

Command and control (C2) is the communication mechanism through which an attacker sends tasks to compromised hosts and receives execution results. Alongside dedicated servers, attackers use **cloud-based and public legitimate services (CPLS)**, including collaboration tools, cloud storage, and developer platforms, to carry this communication. These services expose APIs through trusted domains and protect transport with TLS. Sharing this infrastructure with legitimate activity makes C2 harder to identify through domain reputation or traffic content (Al lelah et al. 2023; MITRE ATT&CK n.d.-a, n.d.-b).

Recent incident investigations further quantify the scale of this abuse. Mandiant's M-Trends 2026 reports that, among the intrusions it investigated in 2025, 15.5% involved Web Service for C2 (T1102), while 5.8% involved Exfiltration Over Web Service (T1567) (Mandiant 2026). Cloudflare's 2026 threat report also documents FrumpyToad conducting C2 by reading and writing encrypted commands in Google Calendar event descriptions (Cloudforce One 2026). These observations show that ordinary platform functions are being used in real attacks, motivating systematic assessment of whether service APIs can be composed into channels that carry tasks and results.

Research on C2 through CPLS has two main lines. Construction studies implement communication prototypes on selected services or service combinations. SLBot combines public services through Service Flux and evaluates its prototype's resilience and efficiency; Alachkar et al. construct channels over six first-party cloud services and evaluate them under enterprise defenses (Alachkar et al. 2026; Wu et al. 2018). Measurement and detection studies instead analyze malware samples, malicious resources, or network records (Allegretta et al. 2025; De Silva et al. 2021; Han et al. 2015; Liao et al. 2016; Yao et al. 2023). Construction studies demonstrate risks on selected services, while abuse analysis generally begins with activity that has already occurred. We ask whether public API documentation alone can support automated assessment of bidirectional C2 feasibility, followed by construction and validation of a corresponding implementation.

This task presents two technical challenges.

Challenge 1: Determining communication feasibility. A service may expose creation, reading, and listing interfaces that operate on unrelated resources, require incompatible permissions, or expose different data to the two endpoints. The task is to determine whether these operations form a single bidirectional communication path. Otherwise, a large language model (LLM) may combine unrelated interfaces into a channel that does not exist.

Challenge 2: Turning a documented communication scheme into a working implementation. Service documentation varies in organization and detail, and some API behavior is ambiguous. Generated code must invoke both service APIs and framework interfaces correctly while handling resource state, failed calls, and repeated interactions. A scheme that appears feasible in documentation may therefore fail when implemented on the real service.

To address these challenges, we present **API2Channel**, an automated workflow for C2 feasibility assessment and implementation generation for CPLS. The system organizes documented resources, operations, and relationships into an attributed resource graph (ARG) with source evidence. It matches topology templates and checks eight communication requirements to identify candidate channels. For the selected candidate, it builds a communication model and an API contract, generates an implementation, and checks and repairs the code using execution feedback in an isolated mock environment. This connects the feasibility decision to a concrete implementation that can be tested. Our contributions are as follows:

- **A unified communication feasibility criterion**: We define candidate channels and eight communication requirements, establishing a common criterion based on whether compatible resources and API operations support task delivery and result return. This criterion provides a basis for documentation-driven assessment without relying on historical malware samples.
- **An automated analysis and modeling pipeline**: We combine API information acquisition, attributed resource graph construction, topology matching, and capability screening to assess feasibility and generate a communication model.
- **End-to-end implementation and isolated validation**: We generate communication implementations from the model and API contract, test them in independently generated mock environments, and use execution feedback to repair errors. This checks whether the documented communication scheme can be implemented.

2 Background and Related Work

CPLS-based C2 reuses legitimate service infrastructure to carry tasks and results. We first review this communication model and the research on its abuse, then explain how mocks support controlled testing of service interactions.

2.1 Command and Control

A C2 channel carries tasks from a controller to an agent on a compromised host and returns execution results. Early botnets often used central servers and Internet Relay Chat (IRC). Fixed servers and channels made them easier to identify and disrupt. Later designs adopted HTTP and HTTPS, fast flux, and domain generation algorithms (DGAs), while peer-to-peer (P2P) architectures reduced dependence on a single control node (Gu et al. 2008; Rossow et al. 2013; Stone-Gross et al. 2009). Fig. 1(a) illustrates an early C2 architecture. These mechanisms improve resilience but typically still rely on attacker-controlled servers, domains, or peer nodes.

Cloud services and web services enable a further shift toward reusing legitimate infrastructure. Their stable APIs, trusted domains, HTTPS transport, persistent resources, and availability can support communication alongside normal activity (Chen et al. 2017; Lehtiö 2015). A third-party service can act as a dead drop resolver (DDR) that publishes the address of a separate controller, or it can carry tasks and results directly. Fig. 1(b) illustrates a C2 architecture using CPLS. We study the latter role: complete bidirectional communication through public service APIs.

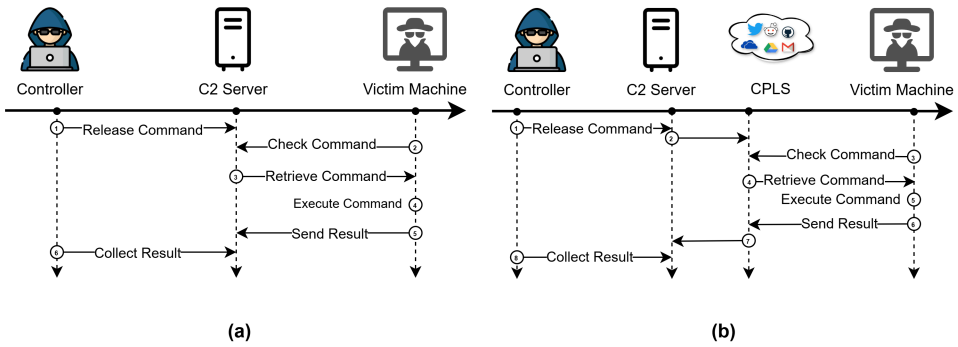


Fig. 1 C2 communication architectures: (a) an early C2 architecture; (b) a C2 architecture using CPLS

2.2 C2 Communication via CPLS

Construction studies demonstrate C2 communication through individual services and combinations of public services (Beckerich et al. 2023; Hilt and Remorin 2017; Janjusevic et al. 2025). Lehtiö discusses web-based C2 using Twitter, Facebook, and Gmail, while CloudBot combines mobile push and cloud storage for command distribution and data exchange (Chen et al. 2017; Lehtiö 2015). SLBot assesses botnet construction using multiple public services, proposes Service Flux with addressing, command, and upload subchannels, and evaluates a prototype's resilience and efficiency (Wu et al. 2018). These studies provide directly relevant precedents for public-service communication construction.

Alachkar et al. (2026) also construct communication prototypes, with an emphasis on enterprise defenses. They develop clients for six first-party services across Azure, AWS, and GCP and evaluate stealthiness and exfiltration performance in an emulated secure enterprise environment. Practitioners from two enterprises each reproduce one scenario in a development environment. These controlled experiments demonstrate how cloud-service channels behave under enterprise defenses.

Abuse measurement and detection studies start from observed malicious artifacts or behavior (Liu et al. 2025). Han et al. (2015) examine malware interactions with public cloud infrastructure, including payload hosting, C2 servers, redirection, and data delivery. Liao et al. (2016) analyze malicious or compromised cloud repositories and use confirmed resources and associated features to detect abuse. De Silva et al. (2021) distinguish public hosting, compromised domains, and attacker-owned domains in malicious URL data to support remediation at different granularities. Marsea reconstructs malware interactions with web applications and supports malicious-content remediation (Yao et al. 2023). More recent measurement documents malware abuse across multiple Internet services (Allegretta et al. 2025).

These studies differ in where their analysis begins. Construction studies select services and implement communication prototypes; abuse analysis starts from malware, malicious URLs, known resources, or observed activity. API2Channel starts

from public API documentation to assess whether the documented capabilities can form a complete channel before malicious evidence is available. When the analysis produces a supported communication model, the system constructs a corresponding implementation.

2.3 Mocking

A mock is a software substitute for an external service in a controlled test environment. A program sends its API requests to the mock, which returns responses and tracks resource state according to predefined rules. For example, a mock file service can save the contents of a creation request and return a file identifier. A later read request can then return those contents. This allows the program to test a sequence of API interactions without contacting the real service.

Mocks provide controlled, reproducible interactions when a web service is unavailable, access is restricted, or its environment is outside the researcher's control. Search-based mock generation can improve test coverage and help discover faults by simulating external service interactions (Seran et al. 2026). For a mock derived from an API specification, request parameters, response fields, and operation semantics define its behavior. The mock is useful only to the extent that this behavior reflects the relevant service capabilities.

API2Channel uses mocks to test whether generated communication code follows its API contract. The contract collects interface definitions and behavioral constraints from public specifications and official documentation. The mock is generated independently from this contract without reading the communication implementation, and its version is fixed before testing. Mock execution checks consistency between the implementation and contract; real-service tests then check whether those interactions work on the actual service.

3 Problem Formulation

To assess different services consistently, we need to specify what a complete channel must do and which access conditions it may assume. We define the participants and threat model first, then the candidate channels and eight communication requirements used throughout the analysis.

3.1 Threat Model

We consider a controller that delivers tasks to an agent, which retrieves those tasks and returns execution results. This setting corresponds to bidirectional communication in MITRE ATT&CK T1102.002 (MITRE ATT&CK n.d.-b). The interfaces and versions listed in Appendix A define the assessment scope for each service. Address-only dead drops, one-way services, and communication that bypasses the public API are outside our scope. The mechanism by which the host becomes compromised is also outside the communication assessment.

The attacker holds an ordinary service account and legitimate API permissions. Both endpoints may use credentials and resources available through documented public procedures. Permissions and quotas are assessed under the basic or default configuration of an ordinary account, without enterprise privileges or specially approved quota increases. The attacker cannot exploit service vulnerabilities, bypass authentication, obtain internal administrative privileges, or control service infrastructure.

3.2 Definitions

Candidate channel. A service may offer several resource combinations for communication, each with different operations and payload fields. We assess each combination separately and call it a candidate channel. A candidate consists of concrete resources, API operations, and payload fields within one service. Downlink tasks and uplink results may use different resources. Their operations must nevertheless be compatible in resource relationships, API versions, and access permissions, so that one endpoint can retrieve data written by the other. All eight requirements below apply to the same candidate.

Communication requirements. To assess whether a candidate channel supports initialization, task delivery, and result return, we organize the required service capabilities into eight requirements. Both endpoints must authenticate, locate communication resources, write payloads, and continuously discover and read data written by the other endpoint. Data must remain available until it is retrieved. These needs correspond to six capabilities: authentication, reading, writing, addressing, persistence, and resource enumeration. Payload capacity and request frequency can still constrain communication when these capabilities are available, so both must also meet the thresholds in the stated evaluation configuration.

Auth. The controller and agent must be able to programmatically invoke all APIs required by the candidate channel under the account and permission assumptions in Sect. 3.1. Their credentials and permission scopes must permit access to the corresponding communication resources.

Read. The agent must be able to read tasks written by the controller, and the controller must be able to read results written by the agent. Reading operations must retrieve the payloads actually written within the candidate channel.

Write. The controller must be able to write tasks through the candidate channel's APIs, and the agent must be able to write results. The written content must be readable by the other endpoint.

Address. During initialization, both endpoints must be able to obtain and use stable identifiers for the agreed communication location and any necessary parent resources. For example, both endpoints can use the identifier of the same Notion page to locate communication resources beneath it.

Persist. Written tasks and results must remain available under the service's documented retention rules long enough for the receiving endpoint to discover and read them during a subsequent access.

Enumerate. For the resource structures assessed here, both endpoints must continuously enumerate task and result resources and obtain the identifiers and routing information needed to retrieve them. Address establishes the initial communication location; Enumerate discovers subsequent communication resources at that location.

Capacity. The raw payload capacity directly supported by the resources used in the candidate channel must meet the minimum specified by the evaluation configuration.

Rate. Under documented rate limits and interface quotas, the candidate must sustain the number of API requests per minute specified by the evaluation profile. When operations have different limits, their request budgets are checked against shared quota scopes and bottlenecks along the communication path. This metric counts API requests; the number required for a complete task–result round trip depends on the interaction process.

The service supplies the basic capabilities; the endpoint protocol defines how to use them for communication. The endpoints can encode agent identities, communication directions, and task–result associations in payloads and interpret them locally. The service therefore does not need native C2-specific functions.

Feasibility criterion. A candidate channel is feasible under the stated access assumptions and evaluation configuration if it satisfies all eight requirements and supports a complete task–result round trip. The controller must write a task; the agent must discover and read it, then write a result; and the controller must discover and read the result and associate it with the task. A service is feasible when at least one such candidate exists. Sect. 4 explains how API2Channel checks these conditions and generates and validates an implementation. This criterion guides documentation-driven predictions. For evaluation, two cybersecurity researchers jointly review all 24 services to establish ground truth, using evidence of complete bidirectional communication under the assessed conditions, including real-service outcomes and observed blocking (Sect. 5.1).

4 System Design

Given a service name, API2Channel applies the criterion in Sect. 3 through three stages: feasibility analysis, code generation, and dynamic validation and repair (Fig. 2). Analysis produces a communication model and an API contract. Generation uses them to build the communication components, which are then tested and repaired using execution feedback.

The communication model describes how the endpoints exchange tasks and results; the API contract specifies the calls and fields used in those interactions. Together, they connect the analysis to implementation and testing. Skills Engineering supplies the workflows, reference implementations, adaptation scripts, and output constraints used across the three stages.

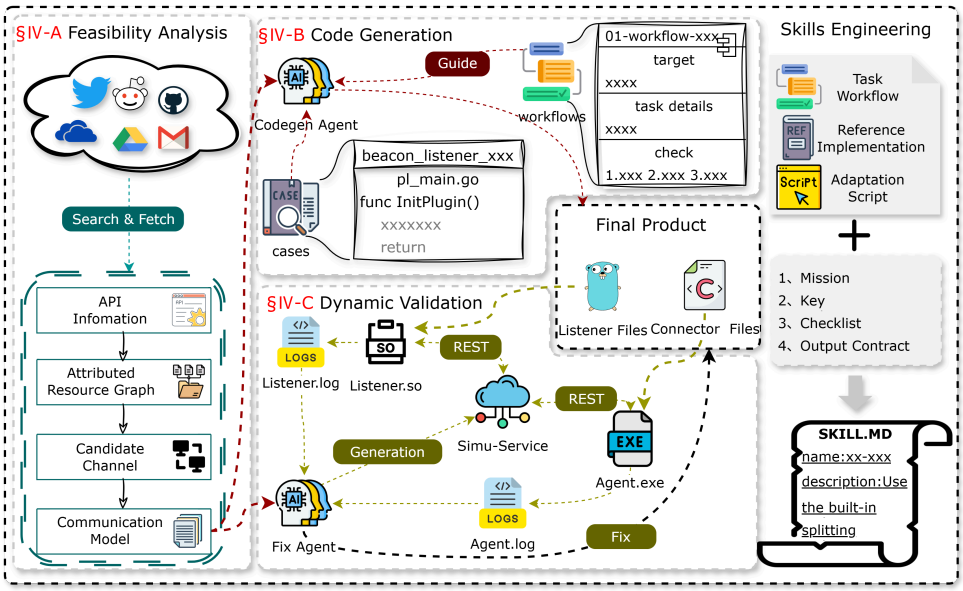


Fig. 2 Overall framework of API2Channel: feasibility analysis, implementation generation, isolated execution and repair, generated components, and workflow guidance

4.1 Feasibility Analysis

Feasibility analysis determines whether documented API operations can form a complete channel. It acquires service information, builds the resource graph, matches topology templates, screens candidate capabilities, and constructs the communication model. Only candidates that satisfy the eight requirements in Sect. 3.2 proceed to modeling. This stage uses public documentation without real-service credentials and produces no communication implementation code.

Information acquisition. Starting from the service name, API2Channel retrieves available structured API descriptions through the APIs.guru Model Context Protocol (MCP) interface and supplements or verifies them using official documentation. Relevant information includes API versions, resource models, authentication, pagination, payload limits, and request quotas. When the catalog does not cover the service or its specification is insufficient, the system consults official documentation directly. The retrieved documents are saved as a fixed local collection for resource abstraction and candidate screening. Users do not need to preselect APIs, resources, or candidate channels.

Resource abstraction. Services differ in their resource names and organization. API2Channel represents them in an attributed resource graph (ARG), allowing the same matching and screening procedures to be used across services. Nodes

represent resources, and edges represent containment, scope, and reference relationships. Resource attributes record API operations, request and response fields, stable identifiers, authentication requirements, operational constraints, and source evidence.

The LLM extracts and normalizes this information from the collected documentation, marking unconfirmed facts as unknown. At this stage, the ARG records service facts; the eight capability values are subsequently determined from the resources, operations, and fields used by each candidate.

Topology matching. To find potential channels in different resource structures, API2Channel matches resources against the four topology templates in Fig. 3. Each template specifies the resource roles and relationships used to distinguish agents and communication directions.

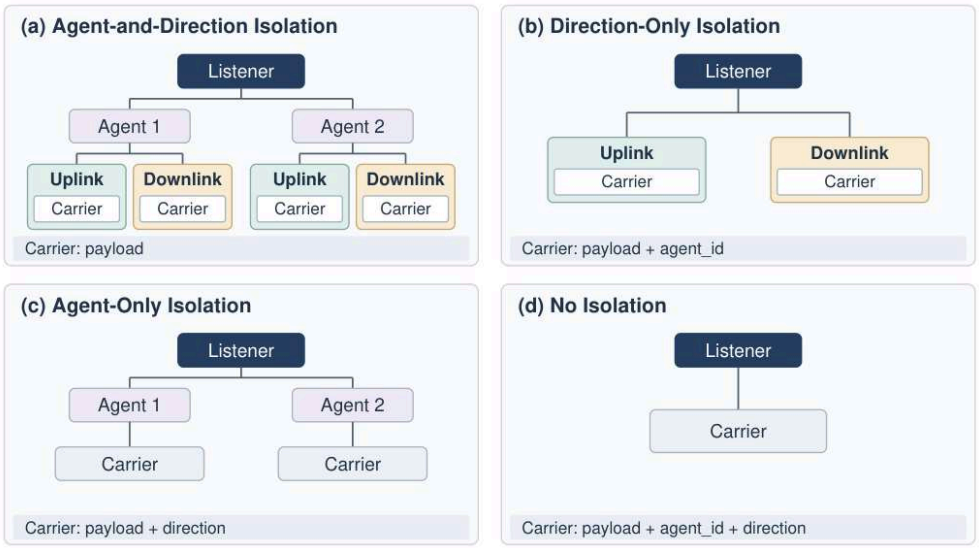


Fig. 3 Four communication topologies: (a) Agent-and-Direction Isolation; (b) Direction-Only Isolation; (c) Agent-Only Isolation; (d) No Isolation. Listener resources distinguish listener instances, and Carrier resources hold payloads. Where resource relations do not separate agents or directions, the payload fields `agent_id` **and** `direction` identify them

Algorithm 1 checks whether a resource-role assignment matches a topology. A template τ specifies roles R_τ, required role traits $P_\tau(r)$, and required containment pairs Q_τ. The assignment μ maps these roles to ARG resource nodes, rather than runtime resource instances. For a node v, $\text{Traits}_G(v)$ denotes the role traits recorded in its attributes A. For each required pair, the algorithm checks for a recorded containment edge supporting child creation, child enumeration, and direct retrieval. Here, E_{contains} is the set of containment edges and $\text{Ops}(e)$ records an edge's supported operations. Missing attributes or relation evidence cause the match to fail.

Capability screening. Algorithm 2 traverses the four topology templates in order, calls Algorithm 1 for each resource-role assignment, and binds compatible API

Algorithm 1 Topology-structure matching (StructureMatch)

Input: ARG $G = (V, E, A)$; template τ; role assignment μ
Output: **true** iff μ satisfies the template's structural constraints

Check resource-role compatibility
if $\mathrm{dom}(\mu) \neq R_\tau$ **then return false**
for $r \in R_\tau$ **do**
  if $\mu(r) \notin V$ **then return false**
  if $P_\tau(r) \not\subseteq \mathrm{Traits}_G(\mu(r))$ **then return false**
Check every required containment relation
$K \leftarrow \{\texttt{create_child}, \texttt{list_children}, \texttt{direct_get}\}$
for $(r, s) \in Q_\tau$ **do**
  $(u, v) \leftarrow (\mu(r), \mu(s))$
  $E_{uv} \leftarrow \{e \in E_{\mathrm{contains}} : \mathrm{endpoints}(e) = (u, v)\}$
  if $E_{uv} = \varnothing$ **then return false**
  if $\nexists e \in E_{uv} : K \subseteq \mathrm{Ops}(e)$ **then return false**
return true

operations and payload fields to form candidate channels. For each candidate χ, the LLM assembles a capability record v from its bound resources, operations, fields, and source evidence. Code then applies the eight requirements in Sect. 3.2 to that record. The algorithm retains only candidates that satisfy every requirement; evidence from other candidates cannot supply a missing capability.

In Algorithm 2, the set $B = \{\mathrm{Auth}, \mathrm{Read}, \mathrm{Write}, \mathrm{Address}, \mathrm{Persist}, \mathrm{Enumerate}\}$ contains the six Boolean capabilities. The values v_C and v_R denote Capacity and Rate, and $\perp$ denotes an unknown numeric value. Capacity is normalized to raw payload bytes after encoding and framing overhead, using the smaller value when uplink and downlink differ. Rate is normalized to API requests per minute under documented default-account quotas, accounting for shared quota scopes and path bottlenecks. The evaluation profile uses $c_{\min} = 1$ KiB (1,024 bytes) and $r_{\min} = 1$ API request per minute; Sect. 5.1 explains their selection. Unsupported or unconfirmed Boolean capabilities and missing or insufficient numeric values cause rejection, with the blocking conditions recorded.

The system selects one of the candidates that pass screening for communication modeling. When multiple candidates pass, a fixed rule first prefers higher Capacity, then higher Rate; remaining ties follow the stable order of the candidate set. Only the selected candidate undergoes complete communication modeling. Passing capability screening alone does not complete the service-level feasibility assessment.

Communication modeling. The selected candidate identifies usable resources and operations, but code generation also needs their interaction order and invocation details. The LLM therefore produces two artifacts from the candidate and a reference modeling document. The communication model describes resource initialization, task delivery, result return, and message correlation. The API contract records the APIs, parameters, response fields, and invocation constraints for those steps. The reference document specifies the required model structure and interaction process.

Algorithm 2 Candidate enumeration and capability screening

Input: ARG G; ordered templates $\mathcal{T}$; thresholds $c_{\min}, r_{\min}$
Output: Candidates $\mathcal{C}$ satisfying all eight requirements, in stable order

$\mathcal{C} \leftarrow \varnothing$
for $\tau \in \mathcal{T}$ **do**
  for $\mu \in \text{RoleAssignments}(G, \tau)$ **do**
    if $\neg\,\text{StructureMatch}(G, \tau, \mu)$ **then continue**
    for $\beta \in \text{CompatibleBindings}(G, \tau, \mu)$ **do**
      $\chi \leftarrow (\tau, \mu, \beta)$
      $v \leftarrow \text{AssessCapabilities}(G, \chi)$
      if $\exists b \in B : v_b \neq$ **true then continue**
      if $v_C = \perp$ **or** $v_R = \perp$ **then continue**
      if $v_C < c_{\min}$ **or** $v_R < r_{\min}$ **then continue**
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{\chi\}$
return $\text{StableOrder}(\mathcal{C})$

The eight capability records constrain the interactions: authentication and addressing support initialization; writing, enumeration, reading, and persistence support task and result delivery; capacity and rate constrain payloads and API invocation schedules. The communication model and API contract use only facts confirmed by the selected candidate and its source evidence.

After modeling, the system reruns candidate matching and capability screening in code. It also checks consistency among the selected candidate, operation contracts, fields used by the model, parameter dependencies, permission scopes, timing constraints, and generated artifacts. The system outputs a feasible decision only when the candidate satisfies all eight requirements and the model specifies the complete task–result round trip defined in Sect. 3.2. The communication model and API contract then pass to code generation.

4.2 Code Generation

Code generation turns the communication model and API contract into two components for the reference framework: the controller-side Listener and the agent-side Connector. The Codegen Agent also receives the guidance, workflows, and cases shown in Fig. 2. Throughout the paper, Listener and Connector name the generated components; controller and agent name the runtime participants.

Code generation. The communication model defines the interaction order, the API contract supplies service-specific calls and fields, and the generation guidance specifies framework structure, component interfaces, and lifecycle constraints. Service facts must come from the published model and contract, rather than being inferred from framework examples.

Generation first produces the Listener, which implements controller-side service interaction, and then the Connector, which implements the corresponding agent-side behavior. Both components use the same resource bindings, operations, and message conventions. The result is a pair of components that the reference framework can load.

Source inspection. The LLM checks interaction steps, resources, fields, directions, and message associations against the communication model and API contract. It corrects implementation errors that can be resolved from the contract before compilation and runtime validation. Missing upstream service facts are recorded for separate reanalysis.

4.3 Dynamic Validation

Dynamic validation checks whether the generated components perform the modeled interaction and uses failures to guide repairs. A shared test harness loads the Listener as a controller-side shared library and integrates the Connector into the agent executable. The endpoints communicate through simulated REST interfaces provided by a mock service, labeled Simu-Service in Fig. 2. The Fix Agent uses runtime feedback from both endpoints to diagnose failures and repair the components.

Mock construction. A separate generation path constructs the mock service from the API contract and mock-generation guidance. It does not read the communication model or generated Listener and Connector source. The mock implements only the interfaces and semantics included in the current contract, including request and response fields, resource relationships, and state changes. This separate generation path reduces the risk of tailoring the test environment to a faulty implementation. Both artifacts can still inherit the same contract error.

The system records the mock's code hash and freezes that version before testing. The mock can be corrected only when recorded runtime evidence demonstrates a conflict with the API contract. A correction creates a new version that is hashed and frozen again; affected prior results are invalidated and their tests rerun. The mock cannot be changed merely to make the current implementation pass.

Isolated execution and validation. The common harness remains fixed across services. It provides the controller with the server interfaces and Listener-loading environment required by AdaptixC2, and provides the agent with the framework source and build interfaces needed to integrate the Connector. Isolated tests validate the coordinated behavior of the Listener and Connector under the same communication model and API contract. The harness provides the same initialization, build, launch, task-delivery, and result-return operations for every service. Service differences are confined to the generated implementation and mock. It collects compiler output, runtime logs, API requests, mock responses, and state transitions. It does not classify services or dynamically alter the test logic. The LLM invokes the test operations in sequence and compares the feedback with the current model and contract. It uses this evidence to relate failures to implementation behavior and service state.

Failure diagnosis and repair. When execution fails, the LLM examines the communication model, API contract, generated source, and harness evidence. It modifies the Listener or Connector to address incorrect calls, field handling, framework adaptation, or state logic. The implementation is recompiled and retested against the same harness and frozen mock. After a local repair, the complete end-to-end sequence is repeated to check for regressions.

The default repair scope contains only the generated communication components. The harness remains fixed, and changes to the mock follow the versioning process above. Execution and repair stop when the implementation passes the prescribed isolated tests or when execution evidence shows that a necessary behavior cannot be implemented under the current API contract. Upstream model or contract defects are recorded and referred for reanalysis.

5 Evaluation

The evaluation asks whether API2Channel makes accurate feasibility predictions, whether its generated implementations work on real services, and how much time and model usage the workflow requires. Sect. 5.1 describes the experimental setup. Sect. 5.2 compares predictions against expert-reviewed ground truth for all 24 services, including the two cases blocked by platform controls. Sect. 5.3 tests the functions of generated implementations on the 12 GT-positive services. Sect. 5.4 reports the time and model usage of the automated workflow.

5.1 Experimental Setup

Evaluated services. Table 1 lists the 24 services, their types, and ground-truth labels. The selection covers services with prior C2 research or open-source implementations, services without reusable public implementations, and clear as well as borderline negative cases. Two cybersecurity researchers with six and eight years of professional experience jointly reviewed all 24 services and assigned their final GT labels. The review determines whether complete bidirectional task–result communication can be established under the assessed interface scope, ordinary-account permissions, default quotas, and test conditions, including payload-capacity and request-rate constraints. Real-service observations, existing implementations, and service-specific evidence support the judgments. The reviewers assigned T to 12 services and F to the remaining 12. Appendix A, Table 4 records the judgments and supporting evidence.

The four core tests cover Listener registration, Agent check-in, heartbeat maintenance, and command execution, confirming task delivery and result return. T and F denote the reviewers' positive and negative judgments of core communication feasibility under the stated conditions. Extended-function failures, such as fragment reassembly, do not change a T label when core communication succeeds. All methods are scored against the same jointly reviewed GT.

Implementation and environment. API2Channel implements its stages as skills. An orchestration skill invokes them in a fixed order and passes artifacts through structured file contracts. The analysis stage uses an LLM for document retrieval, semantic extraction, and modeling, while code performs topology matching and capability checks. The remaining stages generate, inspect, and repair implementations.

We use the same GPT-5.6 Sol (High) model configuration, network access, execution limits, and evaluation date for all assessment methods. The ChatGPT research account used in the experiments had completed OpenAI's cybersecurity researcher verification under Trusted Access for Cyber (TAC) (OpenAI n.d.). Each service is

Table 1 Overview of the 24 evaluated services

No.	Service	Service type	GT	No.	Service	Service type	GT
1	Google Drive	Cloud storage	T	13	Twitter/X	Social media	F
2	Dropbox	Cloud storage	T	14	FCM	Push messaging	F
3	Google Calendar	Calendar	T	15	Pastebin	Text hosting	F
4	GitHub	Developer services	T	16	Weibo API	Social media	F
5	Slack	Messaging	T	17	YouTube Data API	Social media	F
6	Telegram	Messaging	T	18	Discord	Messaging	F
7	arXiv API	Data services	F	19	Notion	Productivity	T
8	OpenWeatherMap	Data services	F	20	Feishu Chat	Messaging	T
9	NewsAPI	Data services	F	21	Feishu Calendar	Calendar	T
10	NVD API	Data services	F	22	Trello	Productivity	T
11	Google Translate API	Language services	F	23	Coda	Productivity	T
12	TextRazor	Language services	F	24	Zulip	Messaging	T

evaluated as an independent task without sharing retrieved documents or decisions. Real-service validation uses AdaptixC2 v1.1, axc2 SDK v1.2.0, Go 1.25.4, and MinGW GCC 13. The server runs in Ubuntu 24.04 LTS under Windows Subsystem for Linux (WSL), and the agent executable runs on Windows 11.

Threshold configuration and rationale. The evaluation profile requires a raw payload capacity of 1 KiB and a rate of 1 API request per minute. These settings represent compact messages and low-frequency asynchronous interaction. The capacity setting draws on tests and message handling in existing manual implementations. C3's ChannelLinter includes a 1,024-byte transfer test, and its Discord text branch limits a packet to 2,000 Base64 characters, or approximately 1,500 raw bytes. OffensiveNotion divides output into 2,000-byte software chunks (OffensiveNotion n.d.; ReversecLabs n.d.). A 1-KiB payload fits below these text or chunk budgets. Encoding 1,024 bytes in Base64 requires 1,368 characters, leaving some space for framing within a 2,000-character carrier. Encoding and metadata overhead must still be checked for each candidate. We treat carrier limits, software chunks, and complete logical messages as different quantities and do not average them.

The rate setting draws on asynchronous interaction at intervals of seconds to minutes. Alachkar et al. (2026) use 15- or 30-second polling intervals in their cloud-service channels, while Cobalt Strike's default Beacon check-in interval is 60 seconds (Fortra n.d.). These examples motivate a low request-rate threshold of 1 API request per minute. They do not directly establish a round-trip rate: polling intervals include waiting and processing, and one interaction may require several API calls. Both thresholds are configurable evaluation settings. Interaction modeling and execution-based validation then check whether tasks and results can complete a round trip.

Baselines. Direct LLM (D-LLM) receives a service name and a request to assess complete bidirectional C2 communication through its public APIs. It retrieves official documentation and returns a feasibility decision, sources, and a short explanation without the eight requirements, ARG, topology matching, or deterministic screening. Direct LLM + Criteria (LLM+) additionally receives the requirement definitions and thresholds and reports capability states and a decision, but still performs direct judgment without the structured analysis algorithms. The baselines let us compare direct judgment, judgment with explicit criteria, and the complete structured workflow.

Functional tests and deployment repair. Second-stage functional testing covers only the 12 GT-positive services. Tests primarily use a single account; messaging implementations typically exchange data through resources such as single-user group conversations. Four core tests check Listener registration, Agent check-in, heartbeat maintenance, and command execution. Three extended tests check file upload/download, multi-agent management, and fragment reassembly. Appendix B gives the settings. In each deployment repair round, a researcher provides failure feedback, the LLM modifies the generated code, and the implementation is redeployed. These rounds measure adaptation needed after isolated validation and are separate from automatic repair within that stage.

Outcome metrics. Generation Success means that an implementation that passed isolated validation can be deployed on the real service without additional researcher-triggered LLM repair. Core E2E Success means that the final implementation passes all four core tests, allowing deployment repair. Full-Feature Success additionally requires all applicable extended tests. These metrics distinguish initial deployment, eventual core functionality, and complete tested functionality. All three second-stage metrics use the 12 GT-positive services as their denominator. First-stage classification metrics cover all 24 services.

5.2 Assessment Accuracy

We first compare API2Channel's documentation-driven feasibility predictions with expert-reviewed ground truth. In Table 2, all three methods identify all 12 positive services. Their differences lie in the 12 negative services: D-LLM produces eight false positives, LLM+ four, and API2Channel two, with no false negatives. API2Channel correctly rejects ten negative services but predicts feasibility for Twitter/X and Discord, both blocked in the tests, yielding 91.67% accuracy. Structured document analysis reduces overprediction of API capabilities but does not yet capture platform abuse controls encountered during execution. We examine these errors below; Table 3 reports the complete classification metrics.

D-LLM's errors show why resource relationships matter. On OpenWeatherMap, it combines measurement submission and weather retrieval into a communication path without confirming that the latter retrieves data written by the former. The model identifies apparent read and write operations, but the claimed payload-transfer relationship is missing.

Giving LLM+ the eight requirements reduces false positives from eight to four, showing that explicit criteria constrain the analysis. The remaining errors show that

Table 2 Communication feasibility decisions on the 24 services. Ours denotes API2Channel. All GT labels are assigned jointly by two cybersecurity researchers. T and F denote positive and negative judgments of core communication feasibility under the stated conditions. All methods are scored against the same GT.

Service	GT	Ours	D-LLM	LLM+	Service	GT	Ours	D-LLM	LLM+
Google Drive	T	T	T	T	Twitter/X	F	T	T	T
Dropbox	T	T	T	T	FCM	F	F	F	F
Google Calendar	T	T	T	T	Pastebin	F	F	T	F
GitHub	T	T	T	T	Weibo API	F	F	T	F
Slack	T	T	T	T	YouTube Data API	F	F	T	F
Telegram	T	T	T	T	Discord	F	T	T	T
arXiv API	F	F	F	F	Notion	T	T	T	T
OpenWeatherMap	F	F	T	F	Feishu Chat	T	T	T	T
NewsAPI	F	F	F	F	Feishu Calendar	T	T	T	T
NVD API	F	F	F	F	Trello	T	T	T	T
Google Translate API	F	F	T	T	Coda	T	T	T	T
TextRazor	F	F	T	T	Zulip	T	T	T	T

the model can still disregard conditions it has identified. For TextRazor, LLM+ reports a request frequency below the threshold used in the recorded run but still returns a feasible label. The error is not simply missing documentation; its final decision contradicts its own capability value.

API2Channel addresses these two errors at different steps. The ARG and candidate bindings constrain which resources and operations can be combined. Code then screens their capability states and numeric thresholds, while the LLM organizes the complete round-trip interaction for the selected candidate. The final prediction must therefore agree with both the candidate's resource relationships and its extracted capability values. API2Channel retains all 12 empirical positives and rejects LLM+'s two additional false positives on Google Translate API and TextRazor, reducing the total from four to two. All three methods still incorrectly predict feasibility for Twitter/X and Discord: capabilities supported by documentation and isolated validation do not ensure communication under dynamic controls on the real service. These results show where structured checks help and where real-service controls still prevent communication.

False positives caused by platform controls. All 14 services predicted feasible by API2Channel complete implementation generation and isolated validation, but only 12 achieve core communication on the real service under the stated conditions. The failures on Twitter/X and Discord demonstrate two direct ways to interrupt communication. Our Twitter/X experimental account is suspended during automated operation, ending subsequent service access. On Discord, Listener registration succeeds but Agent check-in is rejected; the console record attributes the rejection to an Adaptix fingerprint. Account suspension stops further service access; request rejection prevents communication from starting. Both controls stop the generated implementation from completing bidirectional interaction even when the resources and APIs provide the required capabilities. Twitter/X also has one recorded additional deployment repair round without eventual core-communication success. Under the stated GT procedure,

Table 3 Classification metrics against ground truth. Accuracy, precision, recall, and F1 are percentages. TP, FP, TN, and FN denote true positives, false positives, true negatives, and false negatives, respectively.

Method	Acc.	Prec.	Rec.	F1	TP	FP	TN	FN
D-LLM	66.67	60.00	100	75.00	12	8	4	0
LLM+	83.33	75.00	100	85.71	12	4	8	0
API2Channel	91.67	85.71	100	92.31	12	2	10	0

both receive F, and all three methods' T predictions count as false positives. These cases belong to the assessment-error analysis here and are excluded from the functional experiment on GT-positive services in Sect. 5.3.

Successful execution does not mean that a service has no abuse controls. Notion describes automated and manual moderation of public pages and malicious links; Google Calendar processes content for spam filtering and malware protection; Dropbox documents sharing restrictions for suspected malware and bandwidth conditions; and GitHub's acceptable-use policy addresses malicious attack infrastructure and C2 (Dropbox 2026; GitHub n.d.; Google n.d.; Taylor 2021). All four services pass all seven functions in our tests. Detecting malicious files, links, or public content is a different task from recognizing that authorized API reads and writes are being used for C2.

The Twitter/X and Discord cases show how controls can stop the interaction itself. On Discord, fingerprint identification and request rejection prevent Agent check-in; on X, account suspension ends service access. Discord publicly describes malicious-content scanning, and X's automation rules allow account enforcement for prohibited automated activity (Discord n.d.; X n.d.). We do not identify the specific trigger for the X suspension. The observed outcomes nevertheless support linking detection of implementation features or anomalous access behavior (MITRE ATT&CK 2026) to request blocking or account suspension, so that required task and result operations cannot continue.

Finding 1. Combining communication-feature detection with request blocking and account suspension can interrupt C2 channels that are feasible at the API level.

5.3 Real-Service Validation

We next test the generated implementations on the 12 GT-positive services. Fig. 4 shows their deployment repairs, core functions, and extended functions. All 12 complete code generation and isolated validation. Ten need no additional deployment repair; Google Drive and Coda need one and two rounds, respectively. Generation Success is therefore 10/12 (83.33%), Core E2E Success is 12/12 (100%), and Full-Feature Success is 7/12 (58.33%). The 12/12 core result applies to this GT-positive subset; assessment accuracy across all 24 services is reported in Sect. 5.2. Alachkar et al.

(2026) also validate channels through six first-party cloud services in enterprise environments. Our tests focus on the functions and deployment repairs of automatically generated implementations.

Resource selection in a single-account setting. Our real-service tests show that services designed for chat do not have a higher full-function success rate. Four of the six implementations using documents, calendars, or code-hosting text records pass all seven functions. Among the four messaging implementations, only Feishu Chat does so; Slack, Zulip, and Telegram fail fragment reassembly after achieving core communication. The two file-storage implementations, Google Drive and Dropbox, also pass all functions. In these tests, documents and text-storage resources are comparatively easy to use as communication carriers.

The difference relates to how the endpoints use the selected resources. Messaging implementations exchange data through conversations such as single-user group chats. They must locate messages, separate communication directions, and discover later messages. Small per-message limits also require fragmentation and tracking of incomplete payloads. Documents, calendars, and files instead store content in addressable

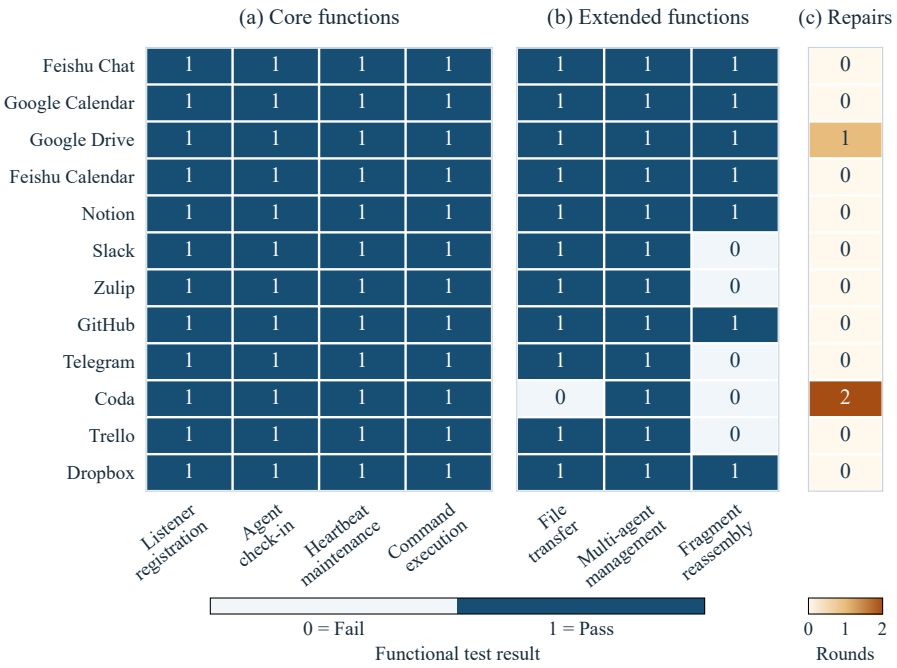


Fig. 4 Second-stage functional validation on the 12 GT-positive services: (a) core functions; (b) extended functions; (c) deployment repair rounds. Every cell shows its observed value. In (a) and (b), 1 denotes pass and 0 denotes failure. Panel (c) uses a separate 0–2 scale for researcher-triggered LLM deployment repairs; larger values indicate more intervention. Darker cells represent larger values within each scale

objects that the endpoints can read and write directly. Fewer resource operations can reduce the state that must be coordinated. This comparison applies to our selected carriers and generated code; a failed test does not rule out a successful alternative implementation.

Finding 2. In our single-account tests, documents, calendars, and similar text-storage resources are easier to repurpose automatically into fully functional C2 channels.

Resource state and implementation errors. When examining additional deployment repairs, we find problems in framework integration and resource-state handling. Integration errors include incorrect use of a framework-provided array. In another case, a resource-creation request times out while waiting for a response. The generated implementation treats the timeout as a failed creation and repeats the request, although the service has already completed the first creation, producing duplicate resources. The interface supports creation, but the program misinterprets the operation's outcome.

The endpoint's local state can then disagree with the service state. An endpoint may treat creation as unsuccessful even though a later read finds the resource; a repeated request can create another copy. The endpoints may consequently use different communication locations or track different positions in the interaction. Notion's documentation describes a related case: a write can return 503 after the change has been saved (Notion n.d.). Validation must therefore check whether an operation took effect and how both endpoints use its result, rather than relying only on the success or failure response.

End-to-end repeatability. To examine whether success depends on a particular generated implementation, we rerun the complete workflow ten times each on Feishu Chat, Google Calendar, and Google Drive. Each run starts from the service name without reusing the previous ARG, communication model, or generated code, and proceeds through analysis, modeling, generation, isolated compilation and repair, and real-service validation. We apply the same success criteria and record the first failing stage and reason if a run terminates early.

Repeated runs reproduce basic communication consistently, while longer interactions still fail. All 30 runs in Fig. 5 pass the four core functions and multi-agent management; file transfer passes in 28 runs and fragment reassembly in 26. The single-run tests show the same pattern: Slack, Zulip, Telegram, Coda, and Trello achieve core communication but fail reassembly, and Coda also fails file transfer.

For sustained communication, both endpoints must agree on which data have been written, which have been read, and when a payload is complete. Fragment handling must preserve this agreement across requests. The duplicate-creation case illustrates how a mistaken local interpretation can cause later interactions to use the wrong resource state. Framework integration is another source of error, and we have not

established the individual cause of every reassembly failure. The observed state mismatch nevertheless explains why correct individual reads and writes are insufficient to ensure correct coordination over time.

Finding 3. Basic task round trips can be constructed repeatedly, while divergent views of shared resource state remain a source of failure in sustained communication.

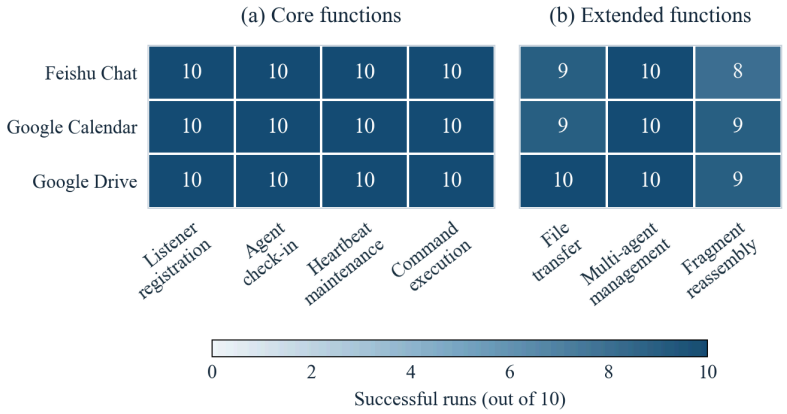


Fig. 5 Functional success counts across ten independent full-workflow runs per service: (a) core functions; (b) extended functions. Every cell gives the number of successful runs out of ten. Both panels use the same 0–10 color scale; darker cells indicate more successful runs

5.4 Cost Analysis

We measure the time and model usage required for feasibility analysis, implementation generation, and isolated validation and repair, excluding real-account configuration, real-service functional validation, and additional deployment repair. The records cover one run for each of the 12 services that eventually achieve core end-to-end communication and ten repeated runs on each of three selected services. They therefore describe resource use for successful constructions and exclude unsuccessful attempts.

Fig. 6 shows execution time and total token usage. The first group contains one run from each of 12 services; the other three groups each contain ten repeated runs on one service. Tables 5 and 6 report token usage by category. Each repeatedly evaluated service contributes one run to the first group as well, so these groups are not fully independent.

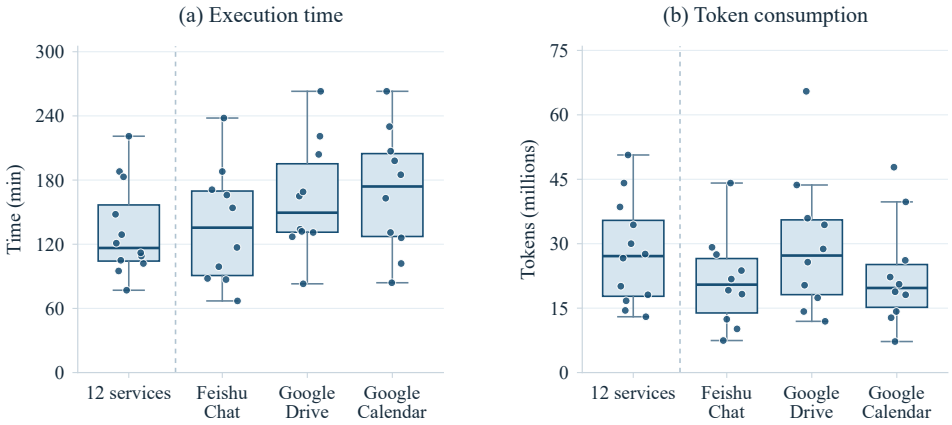


Fig. 6 Execution time (a) and token consumption (b). The first group pools one record from each of 12 evaluated CPLS. The dashed line separates comparisons across evaluated CPLS from repeated runs within individual services. Boxes represent the interquartile range, central lines indicate medians, and whiskers extend to the most extreme values within 1.5 times the interquartile range. Dots show all records

Execution time. The median time is approximately 117 minutes for the single-run group and two to three hours for the three repeated-service groups. The workflow completes service analysis, implementation generation, testing, and automatic repair in hours. Fig. 6(a) also shows variation between runs in the time spent on this workflow.

For a development-time reference, one developer manually implements channels for Feishu Chat, Google Calendar, and Google Drive. The developer has five years of cybersecurity experience and one year of experience using LLMs. This developer-led, LLM-assisted workflow requires approximately two days per service. We record overall development time, without measuring the contribution of LLM assistance at each step. According to the developer's prior experience, comparable C2-channel development typically required weeks before LLMs became widely used; with LLM assistance, it still requires days. The earlier week-scale experience provides context rather than a measurement from this experiment. The manual and automated workflows include different activities in their time totals, and the comparison covers only three services. It provides a descriptive reference for development time, not a controlled estimate of speedup.

Model usage. In Fig. 6(b), median usage is approximately 27 million tokens for the single-run group and 20–27 million tokens for the three repeated-service groups. This usage covers documentation analysis, feasibility assessment, communication modeling, code generation, mock construction, and feedback-driven repair. Token counts measure model use. Monetary cost also depends on the model and the prices for input,

cached input, and output. Input in the tables already includes cached input, and Total equals Input plus Output.

Variation across runs. Execution time and model usage vary with carrier selection, documentation retrieval, and repair. On Google Calendar, for example, generated models may select an event description or properties, leading to different API usage and implementation requirements. Modeling errors can also add repair work or cause construction to fail. These factors vary together, and we do not separately measure their individual contributions to cost.

6 Discussion

The evaluation shows both what the automated workflow can construct and where its predictions remain limited. We discuss how the stages work together, then consider the effects of service coverage, documentation quality, and differences between mock and real-service behavior.

6.1 Implications for Automated Construction

API2Channel links service analysis, communication design, code generation, testing, and repair. Its analysis produces both a feasibility decision and the resource bindings and interaction sequence needed to generate an implementation. All 14 services predicted feasible complete code generation and isolated validation, but platform controls make two of those predictions false positives (Sect. 5.2). Among the 12 GT-positive services included in second-stage testing, all achieve core end-to-end communication and seven pass all seven functions. The results distinguish assessment errors from the functional limits of implementations on services that support core communication.

The eight communication requirements constrain resource selection and interaction design. The communication model and API contract connect analysis results to generated code, while the common harness provides a consistent execution interface across services. Generation, testing, and repair use the same artifacts. This makes the analysis and debugging steps repeatable and reduces the need for repeated developer involvement.

6.2 Limitations and Future Work

Feasibility predictions and ground truth. API2Channel predicts communication feasibility from public API information under the stated access conditions and evaluation profile. The two cybersecurity researchers jointly reviewed all 24 services to determine GT, using open-source reproductions, self-written validation cases, and documentation review. Their judgments concern complete bidirectional communication under the assessed service constraints. Twitter/X and Discord are false positives because real-service controls block their generated implementations. Incomplete documentation or unexplored resource combinations can also cause the system to miss an available channel. The labels apply to the assessed interfaces, accounts, and test conditions.

Service types and documentation dependence. API2Channel currently targets services providing REST APIs and derives resources, operations, permissions, and constraints from public specifications and official documentation. Our resource representation, topology matching, and invocation contracts primarily address these interfaces; we have not systematically evaluated other interaction mechanisms. Missing, incomplete, or outdated documentation can prevent confirmation of required capabilities or produce models that differ from actual behavior. We plan to extend the approach to GraphQL, WebSocket, and event-driven interfaces, adapting resource representation and interaction modeling to subscriptions, push delivery, and persistent connections.

Errors from shared documentation sources. The mock service and communication implementation follow independent generation paths, but both ultimately derive service information from the same document analysis and API contract. Errors in retrieved documentation, semantic extraction, or contract construction can therefore propagate to the communication model, generated code, and simulated environment. For example, the contract may incorrectly state that written data are visible to both endpoints. The mock and implementation can then complete an interaction under the same mistaken assumption, even though it fails on the real service. Passing mock validation primarily establishes consistency with the constructed contract. Independent generation and version freezing reduce the risk of adapting the mock to the implementation but do not eliminate errors inherited from shared information sources. Real-service validation reveals some discrepancies, but these observations have not yet been used systematically to calibrate the mock.

We plan to probe real services automatically using controlled accounts and dedicated test resources, recording requests, responses, and resource-state changes as measured evidence for API contracts and mocks. The probes will cover creation and retrieval, cross-endpoint visibility, identifier stability, pagination, persistence, and error responses. When documentation and observations disagree, we will record the conflict and verify the contract and communication model. Any correction will be followed by mock regeneration or calibration and repetition of the affected validation. Decisions will use actual returned data and observable state, reducing dependence on model-generated expected responses. Account permissions, service versions, and dynamic abuse controls will continue to limit the scope of these observations.

7 Conclusion

We presented API2Channel, an automated workflow that assesses C2 communication feasibility from public API information and constructs corresponding implementations. An ARG, topology matching, and eight communication requirements guide the analysis; a communication model and API contract then guide code generation, testing, and repair. Two cybersecurity researchers jointly reviewed all 24 evaluated services, establishing 12 positive and 12 negative GT labels. API2Channel identifies all 12 positives and correctly rejects ten negatives, achieving 91.67% accuracy. Its two false positives, Twitter/X and Discord, pass isolated validation but are

blocked in the real-service tests. All 14 services predicted feasible complete implementation generation and isolated validation. Second-stage testing covers the 12 GT-positive services: all achieve core end-to-end communication, and seven pass all seven functions. The workflow connects documentation-based assessment to testable implementations, while the observed failures show why API-level capability checks must be complemented by execution on real services.

Ethical Considerations

Stakeholder Analysis. Publishing API2Channel affects service providers, enterprise security teams, security researchers, and ordinary service users. Providers and defenders can use systematic assessment to identify combinations of APIs and resource interactions that may support abuse (OWASP 2023). The results can help them develop monitoring and safeguards. At the same time, automated analysis and implementation generation can lower the expertise and development effort required to construct C2 channels. This could make legitimate services easier to repurpose for malicious communication, exposing enterprise networks, user data, and service reputations to harm. We acknowledge this dual-use risk and weigh the defensive benefits against the possibility of accelerating adversarial adoption.

Mitigations and Notification Status. We use controlled accounts for real-service validation, with implementation testing and repair first performed in an isolated mock environment. The assessment is limited to publicly documented APIs, ordinary account permissions, and basic or default quotas; it does not involve exploiting service vulnerabilities, bypassing authentication, or controlling service infrastructure. We disclosed our findings by email to the providers of every service for which API2Channel generated a channel implementation. No responses had been received as of manuscript submission.

Limited Release Strategy. Reproducibility requires the assessment conditions, decision evidence, communication requirements, functional test settings, and result records. Deployable channel implementations and automated generation capabilities also carry a risk of unauthorized use. Our current release plan is to make only the analysis and validation framework public while restricting the public release of directly deployable channel implementations. This release scope supports examination of the assessment and validation workflow while limiting access to artifacts that can be used directly as operational channels. We retain local snapshots of the research materials and experimental table data. Released materials should exclude sensitive information such as credentials and account identifiers and clearly state the research conditions under which the results apply.

Justification for Publication. Legitimate API operations can be combined into bidirectional C2 communication, while examining individual operations or relying on previously observed abuse offers limited coverage of potential risks across services. This work provides a repeatable assessment workflow and reports both successful real-service communication and false positives caused by platform blocking. These results help defenders understand abusable capabilities and their boundaries under measured

conditions. We publish these findings to help identify and validate potential abuse risks and to inform service governance and monitoring. These benefits depend on testing, disclosure, and release arrangements that match the risks.

Statements and Declarations

Funding. This work was partly supported by the National NSF of China (grants No. 62672193, 62572209) and the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057).

Competing interests. The authors have no relevant financial or non-financial interests to disclose.

Data and code availability. Data supporting this study are currently being organized and are not yet publicly available. We plan to release these data together with the analysis and validation framework once preparation is complete. Public release of directly deployable channel implementations will remain restricted, as described in the Ethical Considerations section.

Research ethics and controlled validation. The findings were disclosed by email to the providers of all services for which channel implementations were generated; no responses had been received as of manuscript submission. See the Ethical Considerations section for the testing scope and release strategy.

Use of AI assistance. We use Codex to assist with English translation and prose editing of the Chinese manuscript. The experimental use of an LLM is described in Sects. 4 and 5.1.

Appendix A Service Evidence Records

Selection rationale. We organize service coverage by the availability of prior communication implementations, then characterize the GT-negative subset by the difficulty of identifying its blocking conditions. These are complementary dimensions: the services without reusable public implementations include both positive and negative cases.

Services with prior implementations (11). Google Drive, Dropbox, Google Calendar, GitHub, Slack, Telegram, Twitter/X, Discord, Notion, Feishu Chat, and Trello have prior C2 cases or open-source channel implementations. They anchor the evaluation in established service-abuse scenarios. We include them to assess whether API2Channel can recover feasible communication structures and construct implementations for previously studied services. Current validation outcomes determine their GT labels.

Services without reusable public implementations (13). This group contains Feishu Calendar, Coda, and Zulip, together with the ten services assessed through documentation review in Table 4. No suitable public communication implementation was available for these services in our assessment. Including them tests whether the workflow can assess services beyond existing reference implementations. Their GT labels include both successful self-written communication cases and negative judgments supported by interface or quota constraints.

Clear and borderline negative cases. The 12 GT-negative services cover two levels of difficulty. Four have direct interface limitations: arXiv API, NewsAPI, and NVD API provide retrieval without a writable shared communication resource, while FCM provides a one-way sending interface without a corresponding result-return path. The other eight require closer examination. OpenWeatherMap and Google Translate accept inputs and return data, but their assessed interfaces do not provide the required shared task–result exchange. TextRazor is limited by payload capacity; Pastebin, Weibo API, and YouTube Data API by default quotas; and Twitter/X and Discord by blocking observed during execution. These cases test whether an assessment can distinguish apparently useful API operations from complete communication under service constraints.

Table 4 records the validation path, outcome or blocking condition, resource scope, and supporting sources for each service. Two cybersecurity researchers with six and eight years of professional experience jointly reviewed all 24 services and assigned their GT labels using three evidence paths. We reproduced available open-source implementations and tested core bidirectional communication on the real services: nine services passed, while Twitter/X and Discord failed. For Feishu Calendar, Coda, and Zulip, no open-source implementation was available for this assessment, so we wrote and executed service-specific cases; all three completed task delivery and result return. For the remaining ten negative services, information collection identified interface or quota conditions that prevented the assessed communication. These activities supply evidence for GT review; Sect. 5.3 separately evaluates implementations generated by API2Channel.

Table 4: Ground-truth validation paths and service-level evidence

Service and decisions	Resources, evidence, and assessed scope
1. Google Drive Cloud storage v3 GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** Folders and files carrying tasks and results. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core bidirectional task–result communication on the real service. The core communication test passed; GT=T. **Sources:** https://github.com/ReversecLabs/C3; https://developers.google.com/workspace/drive/api/guides/limits
2. Dropbox Cloud storage v2 GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** Authorized folder and files. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core bidirectional task–result communication on the real service. The core communication test passed; GT=T. **Sources:** https://github.com/0x09AL/DropboxC2C; https://github.com/ReversecLabs/C3
3. Google Calendar Calendar v3 GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** Calendar and event resources. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core bidirectional task–result communication on the real service. The core communication test passed; GT=T. **Sources:** https://github.com/MrSaighnal/GCR-Google-Calendar-RAT; https://developers.google.com/workspace/calendar/api/guides/quota
4. GitHub Developer services 2026-03-10 (REST) GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** Repository, issue, and comment. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core bidirectional task–result communication on the real service. The core communication test passed; GT=T. **Sources:** https://github.com/ReversecLabs/C3; https://docs.github.com/en/rest/issues/comments
5. Slack Messaging Web API (no separate numeric version) GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** Conversation and message. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core bidirectional task–result communication on the real service. The core communication test passed; GT=T. **Sources:** https://github.com/Coalfire-Research/Slackor; https://docs.slack.dev/reference/methods/conversations.history/

Table 4 continued

Service and decisions	Resources, evidence, and assessed scope
6. Telegram Messaging Bot API 10.3 GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** Bot, chat, and message. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core bidirectional task–result communication on the real service. The core communication test passed; GT=T. **Sources:** https://github.com/3ct0s/dystopia-c2; https://core.telegram.org/bots/api
7. arXiv API Data services Query API (no separate numeric version) GT: F API2Channel: F D-LLM: F; LLM+: F	**Validation basis:** Documentation review **Resources / API scope:** Query requests and Atom entries, excluding website submission. **Outcome / blocking condition:** The assessed Query API retrieves paper entries but provides no means to place tasks or results into a shared communication resource. Website submission is outside the assessed interface scope; GT=F. **Sources:** https://info.arxiv.org/help/api/user-manual.html#311-query-interface
8. OpenWeatherMap Data services Current Weather 2.5; One Call 3.0 GT: F API2Channel: F D-LLM: T; LLM+: F	**Validation basis:** Documentation review **Resources / API scope:** Assessed scope: Current Weather query interfaces and their returned records. **Outcome / blocking condition:** The assessed weather-query interfaces return weather data rather than records that the endpoints can use to exchange their own tasks and results. Measurement submission and weather retrieval do not form the claimed payload-transfer path; GT=F. **Sources:** https://openweathermap.org/api/current
9. NewsAPI Data services v2 GT: F API2Channel: F D-LLM: F; LLM+: F	**Validation basis:** Documentation review **Resources / API scope:** Everything, Top headlines, and Sources article/source resources. **Outcome / blocking condition:** Everything, Top headlines, and Sources retrieve published information but do not let the endpoints exchange their own task and result payloads through the returned article/source resources; GT=F. **Sources:** https://newsapi.org/docs/endpoints
10. NVD API Data services 2.0 GT: F API2Channel: F D-LLM: F; LLM+: F	**Validation basis:** Documentation review **Resources / API scope:** CVE and CPE query records, excluding vulnerability submission and external databases. **Outcome / blocking condition:** The assessed CVE/CPE interfaces retrieve records but do not let ordinary callers use those records to exchange tasks and results. Vulnerability submission and external databases are outside the assessed scope; GT=F. **Sources:** https://nvd.nist.gov/developers/vulnerabilities; https://nvd.nist.gov/developers/products

Table 4 continued

Service and decisions	Resources, evidence, and assessed scope
11. Google Translate API Language services Advanced v3 GT: F API2Channel: F D-LLM: T; LLM+: T	**Validation basis:** Documentation review **Resources / API scope:** Assessed scope: synchronous text-translation interfaces and their requests/responses, excluding glossary-management interfaces. **Outcome / blocking condition:** Synchronous translation returns a translation of the submitted text but does not establish a shared communication location where one endpoint can leave a task or result for the other to retrieve; GT=F. **Sources:** https://docs.cloud.google.com/translate/docs/reference/api-overview
12. TextRazor Language services REST API (no separate numeric version) GT: F API2Channel: F D-LLM: T; LLM+: T	**Validation basis:** Documentation review **Resources / API scope:** Assessed scope: synchronous analysis and a candidate carrying payloads in DictionaryEntry data values. **Outcome / blocking condition:** The assessed DictionaryEntry ASCII/Base64 carrier has an upper bound of 1,000 raw payload bytes, below the required 1 KiB (1,024 bytes); GT=F. **Sources:** https://www.textrazor.com/docs/rest
13. Twitter/X Social media X API v2 GT: F API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** X API access for task and result exchange. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core communication on the real service. The test account was suspended, terminating API access before the task–result exchange completed; GT=F. **Sources:** https://docs.x.com/x-api/posts/create-post
14. FCM Push messaging HTTP v1 GT: F API2Channel: F D-LLM: F; LLM+: F	**Validation basis:** Documentation review **Resources / API scope:** projects.messages.send and destination devices, without combining another service. **Outcome / blocking condition:** The assessed HTTP v1 interface sends messages to destination devices but provides no corresponding return path for the controller to retrieve agent execution results. The send response is not a task result; GT=F. **Sources:** https://firebase.google.com/docs/reference/fcm/rest/v1/projects.messages/send
15. Pastebin Text hosting Developers API (no separate numeric version) GT: F API2Channel: F D-LLM: T; LLM+: F	**Validation basis:** Documentation review **Resources / API scope:** Authenticated user and Paste resources, including creation, listing, raw retrieval, and deletion. **Outcome / blocking condition:** The assessed creation-based path permits 20 new pastes per 24 hours, about 0.0139 creates/minute, and cannot sustain the stated communication workload. The 512-KB paste size is not the limiting factor; GT=F. **Sources:** https://pastebin.com/faq; https://pastebin.com/doc_api

Table 4 continued

Service and decisions	Resources, evidence, and assessed scope
16. Weibo API Social media v2 GT: F API2Channel: F D-LLM: T; LLM+: F	**Validation basis:** Documentation review **Resources / API scope:** User, Status, and Timeline. **Outcome / blocking condition:** Information collection identified an ordinary account's default quota as insufficient to sustain one API request per minute on the assessed communication path; GT=F. **Sources:** https://open.weibo.com/wiki/API%E6%96%87%E6%A1%A3_V2
17. YouTube Data API Social media v3 GT: F API2Channel: F D-LLM: T; LLM+: F	**Validation basis:** Documentation review **Resources / API scope:** Video uploads and CommentThread/Comment resources. **Outcome / blocking condition:** For the assessed creation-based paths, 50 units per comment insertion and 10,000 default units/day permit at most 200 insertions/day (about 0.139/minute). The default 100 video uploads/day equal about 0.0694/minute. These paths cannot sustain 1 request/minute without quota increases; GT=F. **Sources:** https://developers.google.com/youtube/data/determine_quota_cost; https://developers.google.com/youtube/data/reference/rest/v3/commentThreads/insert
18. Discord Messaging v10 GT: F API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** Guild channel and message. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core communication on the real service. Communication initialization was rejected, preventing a complete task–result exchange; GT=F. **Sources:** https://docs.discord.com/developers/resources/message
19. Notion Productivity 2026-03-11 (Notion-Version) GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** A page authorized to both endpoints and its child blocks. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core bidirectional task–result communication on the real service. The core communication test passed; GT=T. **Sources:** https://github.com/mttaggart/OffensiveNotion; https://developers.notion.com/guides/data-apis/working-with-page-content
20. Feishu Chat Messaging IM v1 GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** Chat resources and message exchange. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core bidirectional task–result communication on the real service. The core communication test passed; GT=T. **Sources:** https://github.com/nick-frischkorn/lark_c2; https://open.feishu.cn/document/server-docs/im-v1/message/create

Table 4 continued

Service and decisions	Resources, evidence, and assessed scope
21. Feishu Calendar Calendar Calendar v4 GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Self-written validation case **Resources / API scope:** Calendar and event. **Outcome / blocking condition:** With no open-source implementation available for this assessment, we wrote and executed a service-specific validation case. It completed task delivery and result return on the real service; GT=T. **Sources:** https://open.feishu.cn/document/server-docs/calendar-v4/calendar-event/create; https://open.feishu.cn/document/server-docs/calendar-v4/calendar-event/list; https://github.com/larksuite/oapi-sdk-python/blob/v2_main/lark_oapi/api/calendar/v4/model/calendar_event.py
22. Trello Productivity REST v1 GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Open-source reproduction **Resources / API scope:** Board, list, and card description. **Outcome / blocking condition:** We reproduced an open-source implementation and tested core bidirectional task–result communication on the real service. The core communication test passed; GT=T. **Sources:** https://github.com/gavz/TrelloC2; https://cloud.google.com/blog/topics/threat-intelligence/tracking-apt29-phishing-campaigns
23. Coda Productivity v1 GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Self-written validation case **Resources / API scope:** Document, base table, row, and cell. **Outcome / blocking condition:** With no open-source implementation available for this assessment, we wrote and executed a service-specific validation case. It completed task delivery and result return on the real service; GT=T. **Sources:** https://coda.io/developers/apis/v1
24. Zulip Messaging REST v1 GT: T API2Channel: T D-LLM: T; LLM+: T	**Validation basis:** Self-written validation case **Resources / API scope:** Channel/topic and message. **Outcome / blocking condition:** With no open-source implementation available for this assessment, we wrote and executed a service-specific validation case. It completed task delivery and result return on the real service; GT=T. **Sources:** https://zulip.com/api/rest

Appendix B Functional Test Settings

We consider the core tests successful when the service-specific Listener registers in AdaptixC2, the Agent becomes visible to the controller, heartbeats continue for 24 hours, and the test commands complete and return results. Command tests include `ipconfig`, `mkdir`, `dir`, and deletion.

We test file transfer with a file smaller than the service-specific fragment capacity and use a larger file to trigger fragment reassembly. A transfer succeeds when the file hashes before and after transfer match. File size is adjusted to the selected payload capacity; the exact size for each service is not reported here. We run two agents simultaneously to test whether both execute commands independently. The tests cover the functions listed above.

Appendix C Detailed Token Records

Tables 5 and 6 list our token-usage records by category. Input includes cached input, and Total equals Input plus Output. Some single-run records also appear in the repeated-run table, so the two tables must not be pooled as 42 independent observations.

Table 5: Token counts by category for one run per service

Service	Total	Input	Cached	Output
Feishu Chat	44,122,107	43,929,330	43,293,696	192,777
Google Calendar	18,111,834	18,092,179	17,830,656	19,655
Google Drive	34,395,842	34,313,444	18,894,582	82,398
Notion	12,997,881	12,969,621	12,508,672	28,260
Zulip	50,655,670	50,609,389	49,861,790	46,281
Slack	30,009,394	29,928,797	28,944,733	80,597
Telegram	16,712,904	16,661,567	16,069,633	51,337
Coda	27,599,562	27,516,058	26,614,252	83,504
GitHub	14,466,539	14,432,939	13,932,219	33,600
Trello	20,103,930	20,047,701	19,156,992	56,229
Dropbox	26,636,450	26,558,567	25,804,544	77,883
Feishu Calendar	38,555,227	38,475,644	37,405,158	79,583

Table 6: Token counts by category for repeated runs

Run	Service	Total	Input	Cached	Output
1	Feishu Chat	44,122,107	43,929,330	43,293,696	192,777
2	Feishu Chat	18,263,690	18,105,859	9,741,312	157,831
3	Feishu Chat	27,481,666	27,345,281	24,861,232	136,385
4	Feishu Chat	10,167,585	10,156,005	9,864,960	11,580
5	Feishu Chat	12,416,219	12,355,675	12,023,296	60,544
6	Feishu Chat	23,751,352	23,628,933	21,120,545	122,419
7	Feishu Chat	7,460,661	7,423,635	6,749,271	37,026
8	Feishu Chat	21,774,903	21,666,839	19,698,620	108,064
9	Feishu Chat	29,161,778	29,017,055	26,381,141	144,723
10	Feishu Chat	19,171,013	19,075,872	17,343,017	95,141
1	Google Calendar	14,221,518	14,199,933	13,718,272	21,585
2	Google Calendar	18,818,547	18,734,980	18,346,469	83,567
3	Google Calendar	20,571,022	20,486,631	20,034,432	84,391
4	Google Calendar	12,775,480	12,761,998	12,307,712	13,482
5	Google Calendar	18,111,834	18,092,179	17,830,656	19,655
6	Google Calendar	7,229,671	7,177,499	7,004,800	52,172
7	Google Calendar	26,127,823	26,011,798	25,472,386	116,025
8	Google Calendar	47,813,993	47,659,692	47,019,264	154,301
9	Google Calendar	22,243,688	22,144,911	21,685,687	98,777
10	Google Calendar	39,736,948	39,560,489	38,740,114	176,459
1	Google Drive	14,221,518	14,187,449	7,812,271	34,069
2	Google Drive	65,480,020	65,232,879	55,821,312	247,141
3	Google Drive	28,782,782	28,713,830	15,811,174	68,952
4	Google Drive	11,931,067	11,921,151	11,540,480	9,916
5	Google Drive	34,395,842	34,313,444	18,894,582	82,398
6	Google Drive	25,708,604	25,647,017	14,122,443	61,587
7	Google Drive	20,340,107	20,291,380	11,173,380	48,727
8	Google Drive	17,407,693	17,365,991	9,562,524	41,702
9	Google Drive	43,668,534	43,563,922	23,988,327	104,612
10	Google Drive	35,937,246	35,851,155	19,741,317	86,091

References

Alachkar, K., Barbaro, E., Giuffrida, C., van Eeten, M., Zhauniarovich, Y.: Abusing Cloud Services to Establish Covert Channels in Secure Enterprise Environments. In: 2026 IEEE 11th European Symposium on Security and Privacy (EuroS&P), pp. 753–771. IEEE (2026). https://doi.org/10.1109/EuroSP68448.2026.00054

Allegretta, M., Siracusano, G., González, R., Gramaglia, M., Caballero, J.: Web of Shadows: Investigating Malware Abuse of Internet Services. Computers & Security 149, 104182 (2025) https://doi.org/10.1016/j.cose.2024.104182

Al lelah, T., Theodorakopoulos, G., Reinecke, P., Javed, A., Anthi, E.: Abuse of Cloud-Based and Public Legitimate Services as Command-and-Control Infrastructure: A Systematic Literature Review. Journal of Cybersecurity and Privacy 3(3), 558–590 (2023) https://doi.org/10.3390/jcp3030027

Beckerich, M., Plein, L., Coronado, S.: RatGPT: Turning online LLMs into Proxies for Malware Attacks. arXiv:2308.09183v2 (2023). https://arxiv.org/abs/2308.09183v2

Chen, W., Luo, X., Yin, C., Xiao, B., Au, M.H., Tang, Y.: CloudBot: Advanced Mobile Botnets Using Ubiquitous Cloud Technologies. Pervasive and Mobile Computing 41, 270–285 (2017) https://doi.org/10.1016/j.pmcj.2017.03.007

Cloudforce One: Introducing the 2026 Cloudflare Threat Report. Cloudflare, 3 March 2026 (2026). https://blog.cloudflare.com/2026-threat-report/

De Silva, R., Nabeel, M., Elvitigala, C., Khalil, I., Yu, T., Keppitiyagama, C.: Compromised or Attacker-Owned: A Large Scale Classification and Study of Hosting Domains of Malicious URLs. In: 30th USENIX Security Symposium, pp. 3721–3738 (2021). https://www.usenix.org/conference/usenixsecurity21/presentation/desilva

Discord: Discord Warning System. FAQ on content scanning (n.d.). Accessed 3 November 2026. https://support.discord.com/hc/en-us/articles/18210965981847-Discord-Warning-System

Dropbox: Why has my sharing activity been interrupted? Updated 13 October 2026. Accessed 3 November 2026. https://help.dropbox.com/share/banned-links

Fortra: Cobalt Strike User Guide. Asynchronous and Interactive Operations (n.d.). Accessed 4 November 2026. https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/post-exploitation_asynch-interactive.htm

GitHub: GitHub Acceptable Use Policies. Section 5, “Site Access and Safety” (n.d.). Accessed 3 November 2026. https://docs.github.com/en/site-policy/acceptable-use-policies/github-acceptable-use-policies

Google: Understand the basics of privacy in Google Calendar. Section "Google Calendar uses data to improve your experience" (n.d.). Accessed 3 November 2026. https://support.google.com/calendar/answer/10366125?hl=en

Gu, G., Perdisci, R., Zhang, J., Lee, W.: BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In: 17th USENIX Security Symposium (2008). https://www.usenix.org/conference/17th-usenix-security-symposium/presentation/botminer-clustering-analysis-network-traffic-

Han, X., Kheir, N., Balzarotti, D.: The Role of Cloud Services in Malicious Software: Trends and Insights. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 187–204 (2015). https://doi.org/10.1007/978-3-319-20550-2_10

Hilt, S., Remorin, L.A.: How Cybercriminals Can Abuse Chat Platform APIs as C&C Infrastructures. TrendLabs Research Paper, Trend Micro (2017). https://documents.trendmicro.com/assets/wp/wp-how-cybercriminals-can-abuse-chat-platform-apis-as-cnc-infrastructures.pdf

Janjusevic, S., Baron Garcia, A., Kazerounian, S.: Hiding in the AI Traffic: Abusing MCP for LLM-Powered Agentic Red Teaming. arXiv:2511.15998v2 [cs.CR] (2025). https://arxiv.org/abs/2511.15998v2

Lehtiö, A.: C&C-as-a-Service: Abusing Third-Party Web Services as C&C Channels. In: Virus Bulletin Conference (2015). https://www.virusbulletin.com/conference/vb2015/abstracts/c-amp-c-service-abusing-third-party-web-services-c-amp-c-channels/

Liao, X., Alrwais, S., Yuan, K., Xing, L., Wang, X., Hao, S., Beyah, R.: Lurking Malice in the Cloud: Understanding and Detecting Cloud Repository as a Malicious Service. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1541–1552 (2016). https://doi.org/10.1145/2976749.2978349

Liu, Y., Liu, M., Zhang, Y., Liu, B., Zhang, J., Hong, G., Duan, H., Yang, M.: Dive into the Cloud: Unveiling the (Ab)Usage of Serverless Cloud Function in the Wild. In: Proceedings of the 2025 ACM Internet Measurement Conference (IMC), pp. 63–77. ACM (2025). https://doi.org/10.1145/3730567.3732915

Mandiant: M-Trends 2026. Google Cloud (2026), pp. 7, 99, 101. https://www.gstatic.com/security-marketing/m-trends-2026-en.pdf

MITRE ATT&CK: Suspicious Use of Web Services for C2, DET0425. Accessed 9 October 2026 (2026). https://attack.mitre.org/detectionstrategies/DET0425/

MITRE ATT&CK: Web Service, T1102 (n.d.-a). Accessed 9 October 2026. https://attack.mitre.org/techniques/T1102/

MITRE ATT&CK: Web Service: Bidirectional Communication (T1102.002). Accessed 8 October 2026 (n.d.-b). https://attack.mitre.org/techniques/T1102/002/

Notion: Request limits. Section “Retry a write that returns 503” (n.d.). Accessed 3 November 2026. https://developers.notion.com/reference/request-limits

OffensiveNotion: Source code for Notion-based communication. File: `agent/src/notion.rs` (n.d.). Accessed 4 November 2026. https://github.com/mttaggart/OffensiveNotion/blob/main/agent/src/notion.rs

OpenAI: Models and Trusted Access. Section “Trusted Access for Cyber” (n.d.). Accessed 4 November 2026. https://learn.chatgpt.com/docs/cyber-safety

OWASP: API6:2023 Unrestricted Access to Sensitive Business Flows. Accessed 9 October 2026 (2023). https://api-security.owasp.org/editions/2023/en/0xa6-unrestricted-access-to-sensitive-business-flows/

ReversecLabs: C3 (Custom Command and Control). Source files: ChannelLinter.cpp and Discord.cpp (n.d.). Accessed 4 November 2026. https://github.com/ReversecLabs/C3/blob/master/Src/ChannelLinter/ChannelLinter.cpp; https://github.com/ReversecLabs/C3/blob/master/Src/Common/FSecure/C3/Interfaces/Channels/Discord.cpp

Rossow, C., Andriesse, D., Werner, T., Stone-Gross, B., Plohmann, D., Dietrich, C.J., Bos, H.: SoK: P2PWNED—Modeling and Evaluating the Resilience of Peer-to-Peer Botnets. In: 2013 IEEE Symposium on Security and Privacy, pp. 97–111 (2013). https://doi.org/10.1109/SP.2013.17

Seran, S., Zhang, M., Duman, O., Arcuri, A.: Handling Web Service Interactions in Fuzzing with Search-Based Mock-Generation. ACM Transactions on Software Engineering and Methodology 35(2), Article 51 (2026). https://doi.org/10.1145/3731558

Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., Vigna, G.: Your Botnet Is My Botnet: Analysis of a Botnet Takeover. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 635–647 (2009). https://doi.org/10.1145/1653662.1653738

Taylor, K.: Our content & use policy. Notion, 29 April 2021. Accessed 3 November 2026. https://www.notion.com/blog/our-content-and-use-policy

Wu, D., Fang, B., Yin, J., Zhang, F., Cui, X.: SLBot: A Serverless Botnet Based on Service Flux. In: 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC), pp. 181–188. IEEE (2018). https://doi.org/10.1109/DSC.2018.00034

X: Automation rules (n.d.). Accessed 3 November 2026. https://help.x.com/en/rules-and-policies/x-automation

Yao, M., Fuller, J., Sridhar, R.P., Agarwal, S., Sikder, A.K., Saltaformaggio, B.: Hiding in Plain Sight: An Empirical Study of Web Application Abuse in Malware. In: 32nd USENIX Security Symposium, pp. 6115–6132 (2023). https://www.usenix.org/conference/usenixsecurity23/presentation/yao-mingxuan