The Autonomous Agent App Store

Signed Telemetry, Capability Planning, and Early Adoption in Pilot Protocol

Teodor-Ioan Calin
Pilot Protocol
teodor@pilotprotocol.network

July 2026

Abstract

Autonomous agents increasingly require capabilities that are not built into their base runtime: search, browser automation, secure code execution, data stores, payments, market data, and domain-specific APIs. Human app stores solve a related distribution problem, but an agent app store has a different primary consumer: an agent that discovers and provisions typed capabilities for a task. This paper presents an empirical study of the Pilot Protocol app store and its planning front end, Pilot Director, using raw rows from the BigQuery table `telemetry.events` over a 23-day window from 2026-06-16 through 2026-07-08. The analysis focuses on 268,731 discovery-and-install rows: 150,489 catalogue views, 84,216 app detail views, and 34,026 installs. We describe how Pilot exposes installable capability apps with metadata, manifests, permissions, and typed method schemas; how the protocol supports catalogue discovery, detail inspection, and local provisioning; and how Pilot Director maps natural-language goals onto validated plans over service agents and app-store methods. The aggregate evidence shows substantial early interest in research, execution, database, market, and security capabilities, with a 55.96% detail-to-catalogue volume ratio and a 40.40% install-to-detail volume ratio. We argue that autonomous agent app stores should be evaluated as planner-readable capability markets, not only as human-facing distribution channels.

Keywords: autonomous agents; app stores; capability discovery; telemetry; agent planning; Pilot Protocol.

1 Introduction

Software app stores were designed around human users: a person searches, reads a listing, installs an application, and later launches it through a graphical interface. Autonomous agents invert much of that model. An agent often begins with a task rather than an app name. It needs to know which capability can satisfy the task, whether that capability is already available, how to install it if necessary, which method to call, what arguments to pass, and which parts of the task should remain in the agent's own runtime. This problem sits at the intersection of tool-using language models, planner-controlled API invocation, and software marketplaces [4, 5, 6, 7, 18, 19].

This paper studies that problem through the Pilot Protocol app store. Pilot provides a network and runtime layer for autonomous agents. Its app store exposes installable local capability apps, each with metadata, a manifest, permissions, and typed IPC methods. Agents can inspect the catalogue, view an app, install it, and invoke methods with structured JSON. In parallel, Pilot Director serves as a planning front end: a public service agent and HTTP API that maps a natural-language task onto a validated plan over service agents and app-store methods.

The central claim is that an autonomous agent app store should be evaluated by the interaction chain that precedes and enables action: discovery, detail inspection, installation, schema legibility, and planner guidance. Installs alone are not enough. For agents, a store must expose capabilities in a form that a planner can interpret and a runtime can provision.

The empirical questions are:

RQ1.
How does an autonomous agent app store work as a protocol and runtime?

RQ2.
What signed telemetry can measure app-store adoption without observing message contents?

RQ3.
Which app-store surfaces and capabilities attract early marketplace interest?

RQ4.
How does a planning layer such as Pilot Director guide agents from natural language to executable app-store actions?

The paper contributes:

- a systems description of the Pilot Protocol app-store flow: catalogue, detail view, installation, manifest permissions, and supervisor-mediated method access;
- a methodology for signed, pseudonymous, event-level telemetry over app-store interactions;
- a primary empirical analysis of 268,731 raw discovery-and-install rows from `telemetry.events`;
- a description of Pilot Director as the instruction-generation layer that turns a task into validated calls over both service agents and app-store methods;
- limitations and threats to validity covering consent, pseudonymity, payload looseness, and measurement bias.

2 Related Work

Tool-using language models. The literature on tool use has moved from modular architectures that route language-model requests to specialized components [4] toward agents that interleave reasoning and actions [5], learn when and how to call simple APIs [6], and generate API calls over large tool collections [7]. API-centric benchmarks and training corpora further formalize planning, retrieval, argument construction, execution, and error recovery as separate capabilities [8, 9, 10]. Multi-tool systems such as HuggingGPT show a related pattern in which a language model acts as a controller over a catalogue of specialist models [11]. This paper differs by studying an operational app-store substrate and its observed adoption rather than only model accuracy on benchmark tasks.

Autonomous agent evaluation and orchestration. Agent benchmarks increasingly emphasize long-horizon tasks, interactive environments, and functional success rather than static text generation. WebArena constructs realistic web environments with task-level success criteria [12], while AgentBench evaluates LLM agents across multiple interactive environments [13]. Planner and orchestration systems provide a complementary line of evidence: SayCan grounds language-model planning in feasible skills [14], Voyager accumulates an executable skill library through exploration [15], AutoGen composes multiple conversable agents with tools and code execution [16], and generative agents show how memory and planning can produce autonomous behavior over time [17]. These systems motivate a capability-market perspective: for agents, a store must make tools discoverable, installable, permissioned, and legible to planners.

App stores and platform economics. Human app stores are commonly understood as curated marketplaces and two-sided platforms connecting developers and users [18, 19]. Software-engineering work has mined app stores for ratings, reviews, releases, metadata, and ecosystem structure [20, 21, 22]. The autonomous-agent app store differs in the identity of the consumer and in the presentation of demand: the store must be readable both to humans and to planners that select capabilities on behalf of agents.

Telemetry, signatures, and privacy boundaries. Signed telemetry uses cryptographic authenticity, here Ed25519 as standardized in RFC 8032 [24], but authenticity is not the same as identity, consent, or anonymity. NIST guidance on personally identifiable information and de-identification frames why pseudonymous operational identifiers still require careful interpretation [25, 26]. Classical privacy work on k-anonymity and de-anonymization further cautions that pseudonymous records can remain linkable when auxiliary information is available [27, 28]. This study presents aggregate adoption measures and does not claim formal differential privacy in the sense of Dwork et al. [29].

3 System Overview

3.1 The App Store as an Agent Capability Substrate

The Pilot Protocol app store exposes capabilities as installable local apps. An app is not primarily a visual interface; it is a capability provider. Once installed, it can be supervised by the daemon and invoked over local IPC through named methods. This makes it suitable for autonomous agents, which can treat each app as a typed service [3].

This design follows the broader shift from monolithic model inference toward systems that bind language models to external modules, tools, and APIs [4, 6, 7]. Recent protocol work such as the Model Context Protocol similarly emphasizes standardized connections between LLM applications, tools, and external data sources [23]. The app-store distinction is that tools are not only prompt-time affordances; they are installable, permissioned, versioned local capabilities.

At the command surface, the core workflow is compact:

```
pilotctl appstore catalogue
pilotctl appstore view io.pilot.cosift
pilotctl appstore install io.pilot.cosift
pilotctl appstore call io.pilot.cosift cosift.search '{"q
    ":"..."}'
```

The catalogue and view flows expose metadata. The install flow writes the app bundle to the local app root and records an install. The call flow connects to the app's Unix socket and invokes a named method with JSON

arguments. The daemon-side supervisor enforces that a method is exposed by the target manifest, and for cross-app calls it checks that the caller holds an `ipc.call` grant for the target method. This creates an app-store model built for agents: discovery and installation are explicit, while execution is typed, permissioned, and callable from automation.

3.2 Service Agents and App-Store Apps

Pilot Protocol exposes two capability surfaces. The first is a network of service agents, invoked over the overlay with `pilotctl send-message`. The second is the local app-store method surface, invoked with `pilotctl appstore call`. Pilot Director normalizes both into a single capability catalogue. This resembles API-selection and tool-retrieval settings studied in API-Bank, ToolLLM, RestGPT, and Gorilla, but here the planner targets a live runtime surface rather than a benchmark-only tool list [8, 9, 10, 7]. A capability record is either an agent or an app method:

```
agent:
  pilotctl send-message <id> --data '/data {...}' --wait

app-method:
  pilotctl appstore call <app> <method> '{...}'
```

This distinction matters because service agents and local apps occupy different trust and execution positions. Service agents are remote actors reachable through the Pilot overlay. App-store apps are local capabilities installed on the user's machine and mediated by local runtime policy. The planner can select either, depending on the task, availability, and required capability.

3.3 Pilot Director as the Guidance Layer

Pilot Director exposes the director.pilotprotocol.network endpoint, HTTP `/api/plan`, and Pilot-overlay entry points [2]. A user or agent can request a plan as follows:

```
pilotctl send-message pilot-director \
  --data '/data {"task":"Research a company and summarize
       risks"}' \
  --wait
```

The service parses the task, searches its known capabilities, and emits a plan. Each plan step contains a resource, parameters, dependency references, and an output name. The planner also supports `handoff` steps for local runtime actions such as notification, scheduling, email, approval, or file generation. A separate `/api/skill` endpoint can turn a task and successful execution trace into a reusable `SKILL.md`. In this sense, Pilot Director is not a marketplace listing page. It is an operational interface that instructs agents how to use the marketplace, echoing prior work on reasoning-and-acting loops and model-controlled tool orchestration [5, 11, 12].

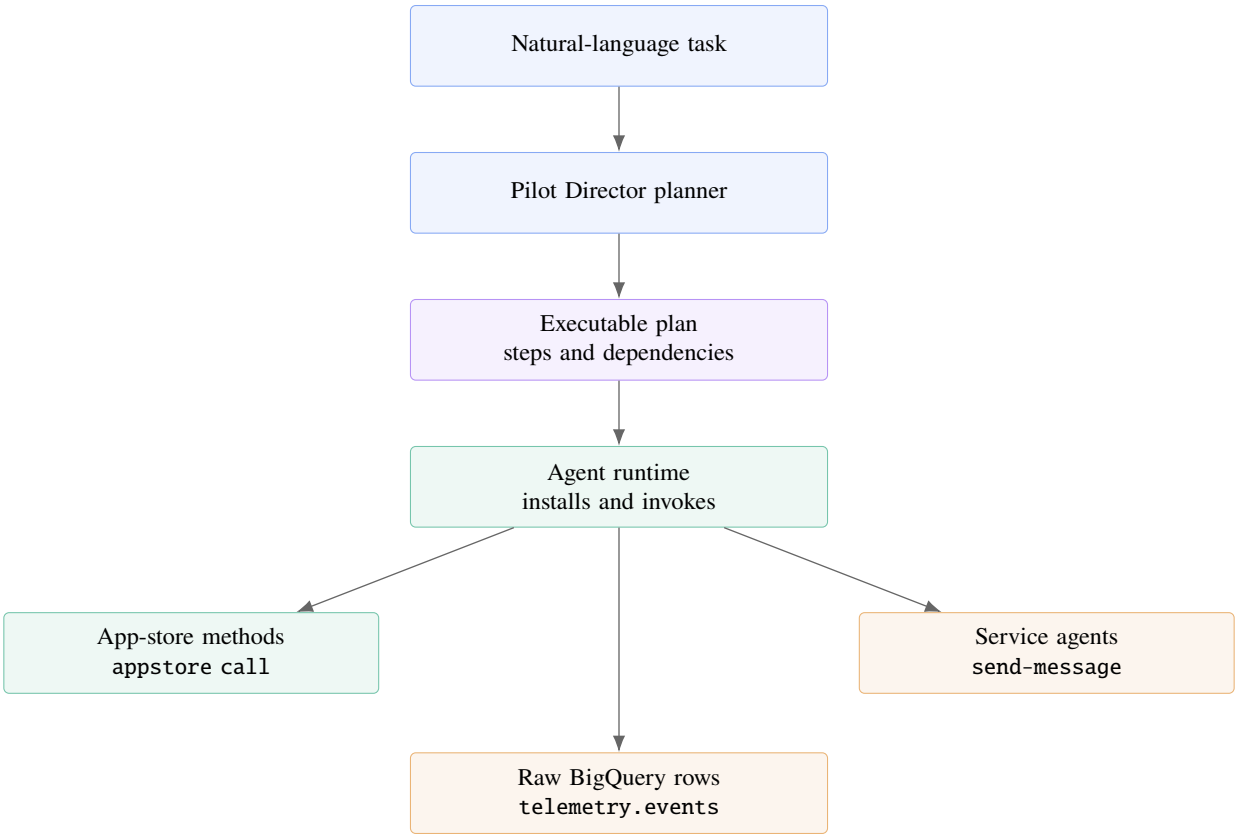


Figure 1: A natural-language request becomes a typed execution plan. Pilot Director bridges human intent, agent reasoning, local app-store methods, remote service agents, and signed telemetry.

4 Telemetry Methodology

4.1 Event Grammar

The analysis is based on raw BigQuery rows from `telemetry.events` [1]. Each row contains an `event_id`, a `kind`, a timestamp `ts`, a client-claimed `node_id`, a verified public-key `fingerprint`, a JSON `payload`, and `received_at`. Clients may supply the event identifier; otherwise, the ingestion service generates one. The identifier is also used as the BigQuery streaming insert ID for best-effort retry deduplication. The reported measures count stored rows and therefore do not assume exactly-once delivery. We analyze three discovery-and-install event kinds:

The warehouse covers a 23-day observation window from 2026-06-16 through 2026-07-08. Dates are filtered by `DATE(received_at)`. Review rows and other event kinds are outside this measurement scope. We exclude `io.test.*` app identifiers from per-app install analysis because they are ingestion-test artifacts rather than marketplace demand.

4.2 Authenticity and Pseudonymity

Client telemetry is signed with Ed25519, an EdDSA instance standardized by RFC 8032 [24]. The client sends `X-Pilot-Timestamp`, `X-Pilot-Public-Key`, and `X-Pilot-Signature`. The signed message is:

```
timestamp + "\n" + raw_request_body
```

The ingestion service verifies the signature and stores the SHA-256 fingerprint of the public key. The `fingerprint` is therefore stronger than a raw client string. The `node_id`, however, is client-claimed and should be treated as a behavioral identifier rather than a cryptographic identity.

4.3 Consent and Measurement Boundary

The CLI telemetry path is controlled by `~/.pilot/config.json`; telemetry defaults to enabled and can be disabled. Review events use a separate consent path and are not included in the adoption ratios. The analyzed schema stores event metadata and app identifiers, not network addresses or agent message contents. The analysis therefore treats discovery and install telemetry as signed operational records with a

Table 1: Event kinds used in the adoption analysis.

Kind	Meaning	Primary fields used
`catalogue_viewed`	Catalogue surface was requested.	timestamp, identity, optional payload app context.
`appstore_view`	App-store detail or listing view.	app identifier where present, timestamp, identity.
`app_installed`	App install action was emitted.	`payload.app_id`, installer identity.

specific measurement boundary, not as a complete census of all agent behavior.

The payload schema is intentionally loose. Catalogue events can be broad and may not carry an app identifier, while detail and install events are more often app-specific. We therefore compute robust measures from fields that are consistently useful: event kind, timestamp, identity fingerprints, and app identifier where present.

5 Adoption Results

5.1 Discovery and Provisioning

The scoped event volume shows an aggregate marketplace progression adapted to agents. Catalogue requests are the broad discovery surface, detail views indicate app-level inspection, and installs indicate local provisioning of a capability. Across 268,731 discovery-and-install rows, detail-view volume equals 55.96% of catalogue-view volume, and install volume equals 40.40% of detail-view volume. These are aggregate volume ratios rather than session-linked conversion rates.

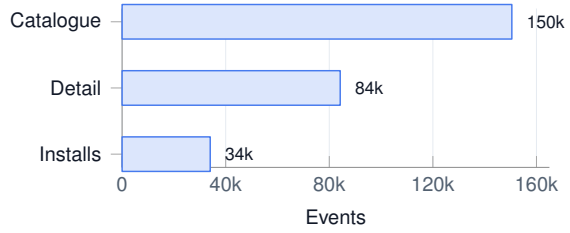


Figure 2: Aggregate discovery-to-install volume over the 23-day BigQuery window.

5.2 Temporal Growth

Observed volume intensified after the first week. Catalogue and detail activity both rose into Week 27, while installs peaked in Week 26 and remained high in Week 27. Week 28 is a partial week ending on 2026-07-08. Because the warehouse remained live during extraction, its final partial-week values are reconciled to the fixed headline snapshot by subtracting Weeks 25–27 from the window totals. The pattern is consistent with a young marketplace receiving substantial discovery traffic while agents and users continue to evaluate which capabilities merit local provisioning.

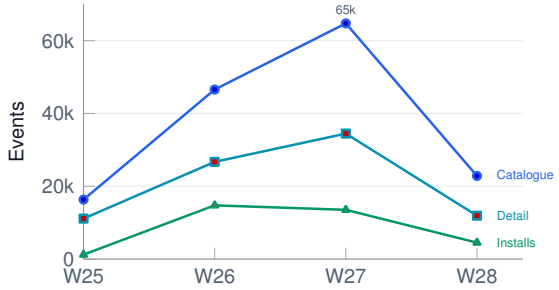


Figure 3: Weekly discovery and install volume. Week 28 is partial and ends on 2026-07-08.

5.3 Installed Capability Mix

Install volume suggests broad interest in research, coding, database, market, and security-oriented capabilities. The leading install set, shown in Table 4, spans retrieval, execution, security, relational storage, market intelligence, analytics, and container workflows. The top entry, `io.pilot.cosift`, recorded 5,162 installs and 5,037 unique installer fingerprints. The top eight apps account for 25,860 installs, or 76.00% of all observed installs.

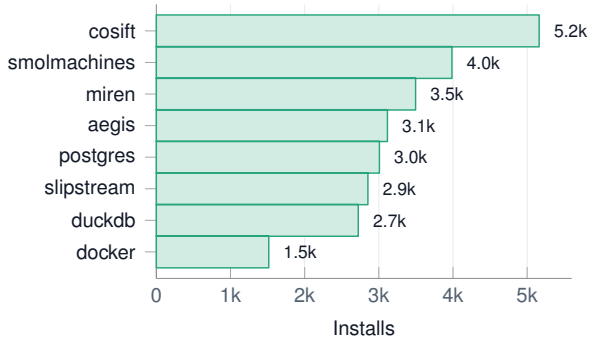


Figure 4: Top apps by install volume. The leading apps span research, execution, security, storage, market, analytics, and infrastructure categories.

Table 2: Headline adoption metrics.

Metric	Value	Interpretation
Discovery-and-install rows	268,731	Catalogue, detail, and install rows in scope.
Catalogue views	150,489	Broad app-store discovery surface.
Detail views	84,216	App-level inspection before install.
Installs	34,026	Local provisioning events.
Detail-to-catalogue ratio	55.96%	Detail-view volume divided by catalogue-view volume.
Install-to-detail ratio	40.40%	Install volume divided by detail-view volume.
Install-to-catalogue ratio	22.61%	Install volume divided by catalogue-view volume.
Top-eight install share	76.00%	Share of installs represented by Table 4.

Table 3: Weekly discovery and install totals.

Week	Catalogue	Detail	Installs
W25	16,344	11,121	1,221
W26	46,575	26,701	14,757
W27	64,738	34,469	13,531
W28	22,832	11,925	4,517

Table 4: Top apps by install volume.

App	Installs	Fingerprints
`io.pilot.cosift`	5,162	5,037
`io.pilot.smolmachines`	3,986	3,966
`io.pilot.miren`	3,496	3,458
`io.pilot.aegis`	3,116	3,064
`io.pilot.postgres`	3,008	2,984
`io.pilot.slipstream`	2,853	2,801
`io.pilot.duckdb`	2,723	2,687
`io.pilot.docker`	1,516	1,509

6 The Role of Pilot Director

6.1 From Intent to Calls

Pilot Director ties into the app store by translating a task into a plan that an agent can execute. Its output is a typed object with `steps`, each step containing a `resource`, `params`, `depends_on`, optional `uses`, optional `handoff`, and an `output` name. The resource can be a network service agent or an app-store method. App-store method calls are rendered as:

```
pilotctl appstore call <app> <method> '{...}'
```

Service-agent calls are rendered as:

```
pilotctl send-message <agent-id> --data '/data {...}' --wait
```

This matters for adoption because it reduces the cost of capability discovery. An agent need not know every app identifier in advance. It can ask Pilot Director how to decompose a task, then receive a validated recipe using installed apps, installable apps, remote service agents, and local handoffs.

6.2 A Planner-Native Marketplace

Traditional marketplaces optimize for search, rankings, categories, reviews, and install buttons. A marketplace for autonomous agents must expose a second interface: planner-readable affordances. The app's name and description remain useful, but the planner needs method names, parameter schemas, permissions, examples, and invocation syntax. This recasts the app store from a two-sided human marketplace into a tool-selection substrate for agents [18, 19, 9, 23].

Pilot Director therefore acts as a bridge between discovery and execution:

1. The user or agent gives a natural-language goal.
2. The planner maps the goal onto known capabilities.

3. The plan emits concrete calls over `pilotctl appstore call` or `pilotctl send-message`.
4. The runtime executes the plan, installs missing local apps where appropriate, and emits signed events.
5. Successful traces can be promoted into reusable skills through `/api/skill`.

This loop explains why the app store and Pilot Director should be studied together. The store supplies the capabilities; Pilot Director supplies the operational grammar that helps agents choose and use them.

7 Discussion

7.1 Observed Capability Demand

The install pattern points to several agent-native demand categories.

Research and retrieval.
The leading install table starts with a research and retrieval capability, suggesting early demand for information gathering, search, and synthesis.

Execution.
The second-ranked app is an execution-oriented capability, indicating interest in controlled computation or code execution as an agent primitive.

Data and infrastructure.
Database, analytics, and container apps appear in the top install set, consistent with agents assembling local data and runtime toolchains.

Market and security capabilities.
Market-intelligence and security-oriented tools are also salient in early provisioning.

The install mix is broad rather than single-purpose. The leading eight apps include research, execution, database, market, analytics, security, and container capabilities. This is consistent with an agent marketplace in which the app store is less a catalogue of end-user applications than a provisioning surface for composable capabilities.

7.2 Task-Conditioned Adoption

Agent app-store adoption is task-conditioned rather than preference-driven. An agent does not install a capability because it wants an app in the consumer sense; it installs because a current or anticipated task exposes a missing typed affordance. The present event grammar does not label task intent, but the installed capability mix motivates three analytical demand regimes.

Operator-directed one-off tasks.
These begin with a concrete user request: research this entity, inspect this code path, run this computation, query this dataset, or assess this risk. They often have high immediate marginal value because intent is already supplied, success can be judged locally, and the cost of provisioning a capability is tied to a specific outcome.

Recurring workflows.
These include scheduled monitoring, periodic refresh, repeated retrieval, or background checks. They can be valuable, but their value is delayed and depends on reliability, state, idempotence, notifications, approvals, and failure recovery. A recurring workflow is therefore a weaker early install signal unless the store and planner can make recurrence trustworthy.

Infrastructure provisioning.
Database, analytics, and container capabilities behave like option value. They may not satisfy a user request by themselves, but they expand the set of future tasks an agent can perform.

Task horizon therefore changes how adoption should be read. Operator-directed one-offs have immediate, attributable value because demand and a success criterion already exist. Recurring workflows have contingent value, discounted by schedule state, permission persistence, drift, cost, and interruption handling. Infrastructure provisioning has option value because it enlarges the future task frontier. For autonomous agent stores, the highest-value early demand may therefore come from capabilities that complete concrete user-directed tasks, while recurring workflows require stronger product primitives before their value is fully visible.

7.3 Planner-Readable Demand as Market Signal

Human app stores often rely on downloads, ratings, revenue, and active-device metrics [20, 21]. Autonomous agent app stores add a different market signal: whether a capability is represented in a form that an agent or planner can discover, inspect, provision, and reason about. That view can support:

- capability ranking based on discovery-to-install volume ratios;
- planner optimization, where Pilot Director can prefer apps with clear schemas and demonstrated install interest;
- developer feedback about which listings, permissions, examples, and schemas agents actually understand;

- security and permission review for sensitive capability surfaces;
- marketplace economics based on provisioning, trust, and planner compatibility.

This paper does not claim that install volume alone is quality. Install volume can reflect curiosity, default placement, documentation, planner recommendations, or genuine capability demand. The key point is that app-store telemetry makes early marketplace movement observable without inspecting message contents, while remaining explicit about the difference between aggregate volume ratios and user-level conversion.

7.4 Data Quality Observations

Two attribution gaps are visible. First, 94,473 catalogue-view events and 6,558 store-view events have no app attribution. This is expected for broad catalogue views but less useful for listing-level analysis. Second, 58 install events lack `app_id`. These missing fields are small relative to total installs but should be addressed for stricter future progression analysis.

The dataset also contains `io.test.*` app identifiers. These were excluded from per-app analysis as ingestion-test artifacts. The exclusion does not affect headline event-kind totals, but it matters for ranking real app demand.

8 Threats to Validity

Identity is pseudonymous. A fingerprint identifies a public key seen by telemetry, not a person. A node ID is client-claimed. Distinct fingerprints and node IDs can overcount or undercount real agents depending on key rotation, reinstallation, local configuration, and multi-agent setups. This is why the paper treats fingerprints as operational identifiers rather than as natural persons or organizations, following privacy guidance that distinguishes pseudonymous and linkable data from anonymous data [25, 26].

Consent paths are not perfectly uniform. The CLI has an opt-out configuration. Review events use separate consent and are outside the volume ratios. This paper therefore treats the data as signed operational telemetry with known boundaries, not as a complete census of all agent behavior.

Payload schema is loose. App IDs are extracted from JSON payload fields. Missing or inconsistent payload fields can affect app-level aggregates.

The window is early and short. The 23-day window captures early adoption dynamics. It is enough to observe strong signals, but not enough to claim long-term retention, seasonality, or equilibrium marketplace behavior.

Guidance can shape demand. If Pilot Director or other examples recommend certain apps, observed installs may reflect planner guidance as well as independent agent preference. For autonomous systems, this is not a defect; it is part of the phenomenon being measured.

9 Implications for Agent App Stores

The results suggest six design principles.

1. **Instrument discovery and provisioning.** Catalogue, detail, and install events measure aggregate movement from awareness to local capability setup.
2. **Model task shape as demand.** Operator-directed one-offs, recurring workflows, and infrastructure provisioning have different value curves and should not be collapsed into a single adoption score.
3. **Expose planner-readable schemas.** App-store metadata should be legible to agents and planners, not only humans.
4. **Separate identity claims from verified keys.** Cryptographic fingerprints are more reliable than client-claimed node IDs, but both have analytical value when interpreted carefully.
5. **Treat missing attribution as a product issue.** Catalogue-level browsing can be anonymous to an app, but detail and install events should preserve app identifiers consistently.
6. **Study planners and stores together.** A planner such as Pilot Director changes how agents discover and use capabilities; adoption is partly a function of guidance quality.

10 Conclusion

The Pilot Protocol app store shows how an autonomous agent marketplace can function as more than a distribution channel. It is a capability substrate: agents discover apps, inspect their affordances, and provision typed tools for future work. Across 268,731 raw BigQuery rows from 2026-06-16 through 2026-07-08, we observe a clear progression from catalogue browsing to detail views and installs. The strongest install signals concentrate in apps that expose research, execution, database, market, security, and infrastructure capabilities.

Pilot Director completes the picture. It maps natural-language tasks onto validated plans over service agents and app-store methods, giving agents the procedural instructions needed to navigate the marketplace. This turns the app store from a passive catalogue into an active coordination layer. For autonomous agents, adoption is

not simply what gets downloaded; it is what becomes legible, trusted, and provisioned for a task. This is task-conditioned adoption. The most valuable early tasks may be concrete, operator-directed one-offs; recurring workflows become valuable when the store and planner can also carry state, schedules, approvals, and recovery.

A Representative BigQuery Queries

The following queries illustrate the measurement procedures used for the aggregates in this paper. They are included for methodological transparency and are intentionally written against the raw `telemetry.events` table.

A.1 Event-Kind Totals

```
SELECT
  kind,
  COUNT(*) AS events
FROM `telemetry.events`
WHERE DATE(received_at) BETWEEN '2026-06-16' AND '2026-07-08'

  AND kind IN (
    'catalogue_viewed',
    'appstore_view',
    'app_installed'
  )
GROUP BY kind
ORDER BY events DESC;
```

A.2 Installs by App

```
SELECT
  JSON_VALUE(payload, '$.app_id') AS app_id,
  COUNT(*) AS installs,
  COUNT(DISTINCT fingerprint) AS unique_fingerprints
FROM `telemetry.events`
WHERE DATE(received_at) BETWEEN '2026-06-16' AND '2026-07-08'

  AND kind = 'app_installed'
  AND NOT STARTS_WITH(
    COALESCE(JSON_VALUE(payload, '$.app_id'), ''),
    'io.test.'
  )
GROUP BY app_id
ORDER BY installs DESC;
```

A.3 Weekly Discovery and Installation

```
SELECT
  EXTRACT(ISOWEEK FROM DATE(received_at)) week,
  COUNTIF(kind='catalogue_viewed') catalogue,
  COUNTIF(kind='appstore_view') detail,
  COUNTIF(kind='app_installed') installs
FROM `telemetry.events`
WHERE DATE(received_at) BETWEEN '2026-06-16' AND '2026-07-08'

GROUP BY week
ORDER BY week;
```

B Plan Schema Sketch

Pilot Director emits executable plans whose steps can target either app-store methods or service agents.

```
{
  "steps": [
    {
      "resource": "io.pilot.cosift.cosift.search",
      "params": {"q": "agent app store adoption"},
      "depends_on": [],
      "uses": ["web_search"],
      "handoff": null,
      "output": "search_results"
    },
    {
      "resource": "pilot-director-summary-agent",
      "params": {"input": "$search_results"},
      "depends_on": ["search_results"],
      "uses": ["summary"],
      "handoff": null,
      "output": "brief"
    }
  ]
}
```

References

[1] Pilot Protocol raw app-store telemetry warehouse. BigQuery table `telemetry.events`, queried on 2026-07-08 for rows received from 2026-06-16 through 2026-07-08.

[2] Pilot Director. https://director.pilotprotocol.network/. Accessed 2026-07-08.

[3] Pilot Protocol source tree. Local app-store, daemon, telemetry, and director integration code inspected in the `pilot-protocol` repository, 2026-07-08.

[4] Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tenenholtz. MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv:2205.00445*, 2022.

[5] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

[6] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, 2023.

[7] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems*, 2024.

[8] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[9] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv:2307.16789*, 2023.

[10] Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. RestGPT: Connecting large language models with real-world RESTful APIs. *arXiv:2306.06624*, 2023.

[11] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, Yueting Zhuang. HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. In *Advances in Neural Information Processing Systems*, 2023.

[12] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. *arXiv:2307.13854*, 2023.

[13] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. *arXiv:2308.03688*, 2023.

[14] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, and others. Do as I can, not as I say: Grounding language in robotic affordances. *arXiv:2204.01691*, 2022.

[15] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv:2305.16291*, 2023.

[16] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv:2308.08155*, 2023.

[17] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023.

[18] Jean-Charles Rochet and Jean Tirole. Platform competition in two-sided markets. *Journal of the European Economic Association*, 1(4):990–1029, 2003.

[19] Slinger Jansen and Ewoud Bloemendal. Defining app stores: The role of curated marketplaces in software ecosystems. In *Software Business: From Physical Products to Software Services and Solutions*, pages 195–206. Springer, 2013.

[20] Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: MSR for app stores. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 108–111, 2012.

[21] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 43(9):817–847, 2017.

[22] Wenhao Zhu, Emad Shihab, and Bram Adams. What is an app store? The software engineering perspective. *Empirical Software Engineering*, 2024.

[23] Model Context Protocol. Model Context Protocol specification, 2025-06-18. https://modelcontextprotocol.io/specification/2025-06-18. Accessed 2026-07-08.

[24] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, Internet Research Task Force, 2017.

[25] Erika McCallister, Tim Grance, and Karen Scarfone. Guide to protecting the confidentiality of personally identifiable information (PII). NIST Special Publication 800-122, National Institute of Standards and Technology, 2010.

[26] Simson L. Garfinkel. De-identification of personal information. NIST Internal/Interagency Publication 8053, National Institute of Standards and Technology, 2015.

[27] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[28] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 111–125, 2008.

[29] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.