Maestro - π

Version: 01
Author : PibyThree Developers

Table of Contents

1. Introduction

This document outlines the requirements, architecture, and implementation details for **Maestro - π** - an in-house, Snowflake-native workflow orchestration platform. The objective of this initiative is to deliver a secure, scalable, and reliable orchestration engine — deployed as **Snowflake Container Services (SPCS)** — that enables organizations to author, schedule, execute, and monitor data pipelines (DAGs) entirely within their own Snowflake environment.

The document serves as a single source of truth for all stakeholders involved in the project, including data engineers, platform/DevOps engineers, solution architects, testers, and project managers. It captures both functional and technical details, along with backend implementation approaches for the ingestion, scheduling, execution, and API layers.

Maestro - π runs as a single Go binary that combines the scheduler, webserver/API, ingestion service, worker pool, and triggerer in one process, backed by a dedicated Postgres metadata store (schema metadata) . It is packaged and distributed as a Snowflake Native App on the Snowflake Marketplace, and runs as two SPCS services (the unified application container + a Postgres container) on a single, app-owned Snowflake compute pool inside the consumer's own account.

This initiative is part of **Pi-Snow** roadmap to offer governed, Snowflake-native workflow orchestration to any Snowflake customer as an installable Marketplace app — without requiring the consumer to stand up or operate an externally hosted orchestrator (Kubernetes/Celery) or a separately managed metadata database themselves.

2. Key Considerations

This section outlines the foundational elements that influence the scope, planning, and execution of the Maestro - π project, including scope boundaries, assumptions, dependencies, and risks.

2.1 Scope

The scope of the project includes the following capabilities:

- DAG authoring in Python , @dag Task Flow decorator, Python Operator/Bash Operator/etc., >>/<< dependencies.
- Git-based DAG ingestion with hash-based change detection and DAG versioning (mid-flight runs pinned to a frozen version).
- Scheduling via cron, named timetables (last_day_of_month, business_days), or dataset-driven triggers (any/all conditions).
- Catchup and historical backfill over a date range.
- Dependency resolution via a directed task graph and a **12-rule trigger-rule engine** (all_success, all_failed , all_done, all_skipped, all_done_setup_success, one_success, one_failed , one_done, none_failed, none_failed_min_one_success, none_skipped, always).
- Dynamic task mapping (.expand()/.partial()) and conditional branching (BranchPythonOperator).
- Typed, JSON-Schema-like run parameters (Param) validated at trigger time.
- Retry policy (fixed or exponential backoff with jitter), execution timeout, and task/DAG-level SLA monitoring.
- Broad operator/connector catalog: Bash, Python/ExternalPython/PythonVirtualenv, Branch, Snowflake, SQLExecuteQuery/MySQL/Postgres/Redshift, S3-to-Redshift, GCS-to-BigQuery, HTTP, SSH, Databricks, Slack, Email, TriggerDagRun, and Sensors (Http/Sql/Time/ExternalTask).
- XCom (cross-task data passing) with Go templating and full Jinja2 templating for Python tasks.
- Alerting via callbacks (on_success/on_failure/on_retry/on_skipped) to Slack, Email (SMTP), PagerDuty, or a generic HTTP webhook.
- Centralized monitoring: dashboard, grid (run x task matrix), per-try log viewer, dead-letter queue, landing-time (SLA) analytics, and real-time task-state updates over WebSocket.
- Role-based access control (RBAC) plus per-DAG resource-level permissions.
- Execution entirely within the Snowflake SPCS framework, packaged and distributed as a Snowflake Native App — Postgres as the sole coordination plane (no external queue/cache).
- Packaging and distribution as an installable Snowflake Native App (Application Package) — automated compute-pool/warehouse/service provisioning via manifest-driven lifecycle callbacks once the consumer approves privileges and binds their own External Access Integration.

2.2 Out of Scope

The following items are not included in the current scope:

- Automated correction, modification, or deletion of source/target data (task authors are responsible for their own operator logic).

- Data-quality profiling, rule-based data assertions, or anomaly scoring on dataset contents (this is a distinct product concern, not part of Maestro - π's orchestration layer).
- Provider-operated hosting of a consumer's installed app — Maestro-π Postgres, generated secrets, and DAG execution state live entirely inside the consumer's own Snowflake account; πby3 does not run or access a consumer's live instance (this is an installable package, not a πby3-hosted SaaS).
- Automatic infrastructure autoscaling — scaling the compute pool or service instance count is a manual ALTER SERVICE / ALTER COMPUTE POOL operation (or an external automation reading the exposed ScheduledTasksWaiting Prometheus metric); Maestro - π does not call the Snowflake API to scale itself.
- A managed/external metadata database — the metadata store is an app-managed Postgres container service running inside the consumer's own Snowflake account, not a separately provisioned Snowflake-managed Postgres instance or a database hosted outside the consumer's account.

2.3 Assumptions

The successful installation and operation of Maestro - π, as an installed Snowflake Native App, is based on the following assumptions:

Sr. No.	Description	Implications if Not Met
1	Snowflake account with Snowpark Container Services (SPCS) enabled and permitted to install Native Apps (Marketplace listings).	Application cannot be deployed or executed.
2	The installing role (ACCOUNTADMIN or a delegate) approves the app's requested privileges (CREATE COMPUTE POOL, CREATE WAREHOUSE, BIND SERVICE ENDPOINT, IMPORTED PRIVILEGES ON SNOWFLAKE DB) and binds a consumer-owned External Access Integration.	App cannot auto-provision its compute pool, warehouse, or services — creation stays deferred until approved.
3	The consumer creates and owns a network rule + External Access Integration covering GitHub, Slack, and *.snowflakecomputing.com, then binds it to the app's Maestro_Pi_eai_ref reference.	Git ingestion, Slack alerts, and Snowflake-operator tasks fail.
4	Sufficient compute pool resources (the app-owned MAESTRO_PI_POOL) are available for the worker goroutine pool and any local (Bash/Python) task subprocesses.	Tasks may queue, time out, or degrade in performance.
5	DAG authors are trusted; Git-repo access to the dags/ subdirectory is itself governed outside Maestro - π (e.g., repo branch protections).	Untrusted DAG code executes with full parser builtins during ingestion.

2.4 Dependencies

Sr. No.	Description	Owner	Implications if Not Met
1	Automated provisioning of the SPCS compute pool and the two services (app + Postgres) once the requested privileges are approved and the EAI is bound.	Maestro-π app (automated) — triggered by the consumer's Account Administrator approving privileges and binding the EAI	Application cannot be installed or run.
2	Network rule + External Access Integration covering GitHub, Slack, and Snowflake hosts, created and owned by the consumer, then bound to the app's Maestro_Pi_eai_ref reference.	Consumer's Account Administrator	Ingestion, alerting, and Snowflake-operator tasks are unavailable.
3	Boot-critical secrets (Postgres password, JWT secret, connection-encryption AES key) are generated once by the setup script inside the installed app and persisted in the app's own config schema — no consumer secrets file required.	Maestro-π app (automated at install)	N/A for boot-critical secrets (generated automatically); if optional SMTP settings are left blank, email alerting is simply disabled.
4	Git repository reachability (HTTPS) through the bound External Access Integration; for private repositories, credentials must be available to the container.	Consumer (DAG repository owner)	DAG ingestion cannot fetch source files.
5	Postgres service availability (block volume, correct schema application), created and kept running automatically by the app.	Maestro-π app (automated)	No coordination plane — nothing in the app can run (scheduling, locking, state).

2.5 Risks

Sr. No.	Description	Mitigation Strategy
1	The app-owned compute pool is capped at MIN_NODES=1/MAX_NODES=1 in the packaged app — throughput ceiling under heavy load.	Scale via ALTER COMPUTE POOL MAESTRO_PI_POOL SET MAX_NODES=n / ALTER SERVICE ... SET MIN_INSTANCES=n run by the consumer's own account admin.
2	Leader crash halts scheduling until failover (~30s lock-staleness threshold).	Run >=2 instances so a worker can auto-promote to leader; monitor scheduler_lock.heartbeat_at.
3	CPU-heavy local tasks (Bash/Python subprocesses) can contend for a single instance's CPU.	Admission control (MaxLocalTasks cap, Elevated/Critical load shedding) plus spreading instances across nodes.

3. Business Feature Details

This section describes the key business features of Maestro - π, including DAG configuration and execution, access management, monitoring, and alerting capabilities.

3.1 Business Process Flow

Maestro - π enables organizations to author, schedule, execute, and monitor data pipelines through a configurable and governed workflow.

The business process consists of the following steps:

1. Author writes DAG as Python files in a Git repository under a dags/ subdirectory.
2. Ingestion polls/fetches the repo, hash-diffs changed files, and parses them via a Python subprocess.
3. Parsed DAGs (tasks, dependencies, params, schedule) are persisted to Postgres and versioned.
4. Configure DAG-level settings (schedule, timetable, catchup, concurrency limits, run timeout, DAG-level SLA, typed params, callbacks) and task-level settings (retries, trigger rule, pool, priority weight, execution timeout, task SLA) directly in DAG code.
5. The scheduler creates DAG runs (cron/timetable/dataset-triggered or manual trigger with a custom conf) and plans/expands tasks per dependency and trigger-rule evaluation.
6. The worker pool (leader + worker nodes) claims and executes tasks via type-dispatched executors.
7. Review execution results via the dashboard, grid view, per-try logs, landing-time analytics, and the dead-letter queue.
8. Alerts fire via configured callbacks (Slack, Email, PagerDuty, or webhook) on task/DAG success, failure, retry, or skip.
9. RBAC and per-DAG permissions govern who can view, trigger, edit, or clear which DAGs.

3.2 RBAC and UBAC

As a Native App, reaching Maestro - π at all first requires the Snowflake-level Maestro_pi_user application role (granted by the consumer's account administrator); the installer is auto-provisioned as the first in-app Admin (bootstrap_admin). Beyond that gate, Maestro - π protects access using two complementary controls:

1. Roles determine what a user can do

- Users are assigned one or more of five seeded roles: **Admin**, **Op**, **Editor**, **Viewer**, **Public**.
- **Admin** — full management rights: */*/* — configure connections, manage users/roles, administer the system.
- **Op** — view/trigger DAGs & runs, delete runs, view/clear/mark tasks, view connections/variables, view+edit pools, view system.
- **Editor** — Op permissions + edit DAGs, edit/delete connections & variables (no admin/system-edit).
- **Viewer** — read-only access to DAGs, runs, and tasks.

- **Public** — unauthenticated read-only access to DAGs/runs/tasks, only if PUBLIC_READ_ENABLED is set.
- Permission grants are modeled as (role_id, resource_type, resource_id, action), resource types dag/connection/variable/pool/system/dag_run/task/*
- Actions: view/edit/trigger/delete/clear/mark/admin/*.

2. Resource-level controls (per-DAG permission — the UBAC-equivalent) determine what a user can see and act on

- Per-DAG ACLs live in the dag_permission table: can_read / can_trigger / can_edit / can_delete / can_clear, per (dag_id, role_id).
- source='dag_file' — declared in DAG code and replaced wholesale on each ingestion.
- source='admin' — set via the API/UI and preserved across ingestion (not overwritten by the DAG file's declared permissions).
- This means a user may hold an application role (e.g., Viewer) but still be denied access to a specific DAG, or conversely be granted elevated per-DAG rights beyond their base role.

3.3 DAG Configuration & Execution

Maestro - π enables users to configure and execute DAGs across a single ingested Git repository, all DAGs share one Postgres metadata store.

Users define scheduling and execution behavior at the **DAG** level and the **task** level. The platform supports:

- **Automatic runs** from cron / named timetable / dataset-driven triggers.
- **Manual trigger** with a custom conf, validated against the DAG's typed params schema.
- **Backfill** over a historical date range, honoring or bypassing catchup.
- **Cross-DAG triggering** via TriggerDagRunOperator (conf, wait_for_completion, allowed_states, failed_states).
- **Dynamic task mapping** — fan-out via .expand() (literal list or upstream XCom) + .partial(), with per-slice mapped_params.
- **Branching** — BranchPythonOperator selects which downstream task_id(s) run; others skip.
- **Sensors** — Http/Sql/Time/ExternalTask, in poke mode (holds the slot) or reschedule mode (frees it between pokes), plus deferrable/async sensors via the triggerer.

Run/task configuration spans:

- Concurrency: max_active_runs, max_active_tasks, per-task task_concurrency, pools with slot limits.
- Reliability: dagrun_timeout_seconds, task execution_timeout_seconds, retries with fixed or exponential backoff + jitter.
- Prioritization: priority_weight + weight_rule (absolute/upstream/downstream).
- SLA: DAG-level expected_duration_seconds and task-level sla_seconds.

Run and task-instance results are stored in Postgres and made available through the dashboard, grid view, historical trend views (landing times), and per-try logs.

3.4 Monitoring & Reporting

Maestro - π provides centralized monitoring capabilities to enable continuous visibility into pipeline execution. The platform offers:

- Aggregate dashboard stats (run/task success counts, etc.).
- Grid view — a run x task status matrix.
- Per-try task log viewer.
- Dead-letter queue (DLQ) viewer for terminally failed tasks.
- Landing-time / SLA analytics for timeliness trend review.
- Real-time task-state push over WebSocket (no manual refresh needed).
- Manual "sync now" trigger for on-demand DAG (re-)ingestion from Git.
- Audit log of all mutating API actions (admin-facing).

4. Technical Feature Details

This section outlines the technical interpretation of the business features above, translating them into system components.

4.1 Technical Architecture

The solution runs as a single Go binary with a containerized Postgres container, packaged as a Snowflake Native App and deployed as two SPCS services on one app-owned Snowflake compute pool inside the consumer's account — provisioning is automated by the app's setup script and lifecycle callbacks:

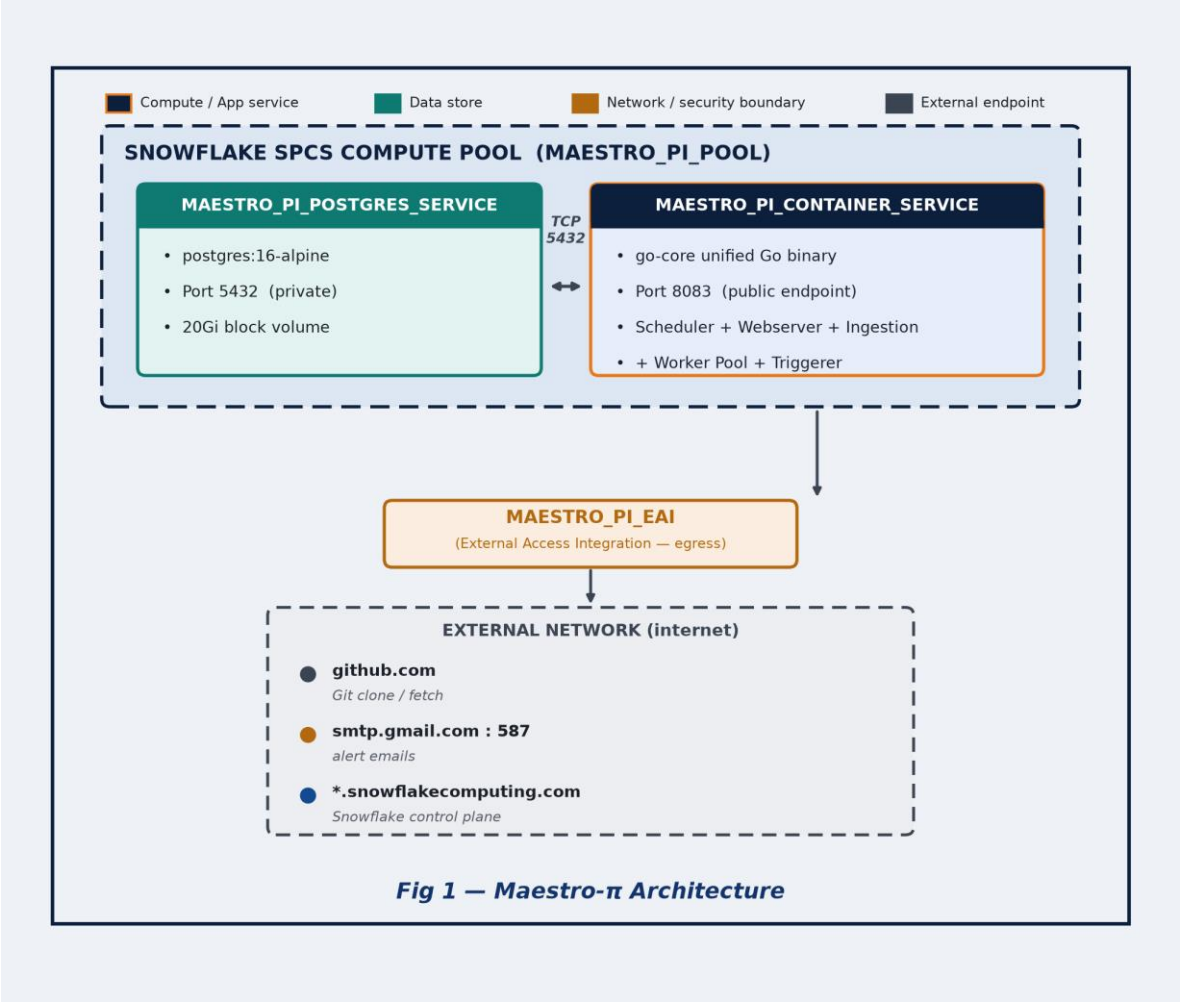

Fig 1 — Maestro-π Architecture

Key components:

- **Webserver / API** — Gin REST API + WebSocket, port 8083, on **all** node roles.
- **Scheduler** — DAG-run creation, task planning/dispatch, reliability subsystems; **leader only**.
- **Ingestion** — Git sync + Python-subprocess DAG parsing + persistence; **leader only**.
- **Triggerer** — polls deferred trigger_instance rows and fires them; **leader only**.

- **Worker Pool** — autoscaling goroutine pool executing task instances; **all** node roles (local channel dispatch on the leader, FOR UPDATE SKIP LOCKED claim on worker nodes).
- **Postgres metadata repository** — the *sole* coordination plane (schema metadata): DAGs/tasks/runs/task_instances, RBAC/dag_permission, connections, variables, scheduler lock, worker_node registry, DLQ, audit log, etc. No message queue, cache, or external lock service.
- **Snowflake Data Layer** — reached only by task-level connectors (e.g., the Snowflake operator), not by Maestro - π's own state management.

4.2 Scheduler & Execution Engine

The Maestro - π scheduler and worker pool together form the orchestration engine responsible for turning parsed DAG definitions into executed task instances. Key capabilities include:

- Cron / timetable / dataset-triggered DAG-run creation, with catchup and per-DAG guardrails (max_active_runs, end_date).
- Task planning from a frozen dag_version snapshot (so mid-flight runs are unaffected by DAG edits), with priority-weight computation.
- Dynamic task expansion (.expand()) and branch-decision application.
- A 12-rule trigger-rule engine evaluating upstream state counts to decide readiness.
- Adaptive-batch dispatch to a local worker-pool channel (leader) with overflow claimed by remote workers via FOR UPDATE SKIP LOCKED, woken by pg_notify('task_scheduled').
- A deferred-task triggerer for async sensors/operators, freeing worker slots while waiting.
- Reliability subsystems: zombie detection (stale-heartbeat recovery), stale-run cleanup, retry/sensor rescheduling, SLA-miss monitoring, XCom retention cleanup.

The engine executes task instances against user-selected operators/connectors and stores the resulting state, logs, and XCom values for subsequent analysis and monitoring.

4.3 Connections & DAG Configuration

Maestro - π enables users to register reusable external connections and configure DAG-level settings, all centrally through the metadata store (a single Postgres instance backs the whole deployment — there is no separate "account" onboarding step the way a multi-tenant scanning product might have one per target account).

Connections And Operators

- **Connector And Operators Currently present** : Snowflake, Slack, BashOperator, EmailOperator, BranchPythonOperator, HttpSensor, TriggerDagRunOperator
- **Connectors Planned to be added** : Postgres, MySQL, RedShift, GCS, BigQuery, AWS, SSH.
- CRUD + a live **Test Connection** probe per type (e.g., SELECT 1, list bucket).
- Secret fields AES-256-GCM encrypted at rest (CONNECTION_PASSWORD_AES_KEY).

DAG Configuration (declared in DAG code, persisted at ingestion time)

- Identity/docs: dag_id, description, owners, tags.
- Schedule: cron schedule_interval, named timetable, or dataset-driven schedule=[Dataset(...)].
- Time window: start_date, end_date, per-DAG timezone.
- Catchup/backfill toggle, run/task concurrency limits, run timeout, DAG-level SLA.
- Typed params schema (JSON-Schema-like), validated at trigger time.
- DAG-level callbacks, Jinja customization (user_defined_macros/filters, native-object rendering).
- Partition definitions for partition-scoped runs.

Rule/Trigger Assignment (task-level, declared in code)

- Per-task trigger rule (one of 12), retry policy, pool, priority weight, SLA, timeout, setup/teardown role, environment overrides (env=, python=, venv=).

4.4 Security & Access Control

Maestro - π implements Role-Based Access Control (RBAC) plus per-DAG resource permissions (see Section 3.2) to provide secure, governed access to the application and its DAGs.

Auth model

- Snowflake-native identity: in the packaged app, the SPCS ingress injects a trusted caller identity that Maestro - π maps to its own Maestro_pi_user records — there is no Maestro - π login screen or password store; the session/JWT/API-key mechanisms below layer application-level session management on top of that Snowflake-verified identity.
- Priority chain: session cookie -> Bearer JWT -> X-API-Key header -> public role (if enabled) -> 401.
- JWT: HS256, JWT_SECRET, access token default 15 min, refresh token default 7 days; claims = UserID, Username, Roles[].
- Sessions: cookie-based, stored in Maestro_Pi_session, instantly revocable.
- API keys: X-API-Key header, stored hashed, up to API_KEY_MAX_PER_USER per user.
- Connection secrets: AES-256-GCM at rest.

Relationship with Snowflake / target-system permissions

Maestro - π's RBAC and per-DAG permissions govern application functionality and DAG visibility *within Maestro - π*. They do **not** grant privileges on any target system a task connects to. The Connection configured for a task (e.g., the Snowflake connector) must independently hold the required privileges on the target warehouse/database/schema/table — if those are missing, the *task* fails at execution time even though the *user* was authorized to trigger it.

5. Technical Details

5.1 Snowflake Container Services (SPCS) Deployment

This section describes how Maestro - π is deployed using Snowpark Container Services as a packaged Snowflake Native App. Maestro - π is distributed via an Application Package and installed by a consumer through the Snowflake Marketplace — there is no manual SQL to hand-run for compute-pool or service creation; the app's own setup script and lifecycle callbacks provision everything once the consumer approves privileges and binds an External Access Integration.

- Maestro - π runs as two Snowflake services on a single, dedicated compute pool inside your own Snowflake account — the unified application service, and a private Postgres service that stores orchestration state.
- Both services, the compute pool, and a dedicated warehouse are created automatically once you approve the requested privileges and bind your External Access Integration — there is nothing for you to build, push, or hand-run.
- The application is reachable through a Snowflake-managed public web endpoint (the UI and API); the Postgres service stays private and is never reachable from outside your account.
- Settings you can change — such as the Git repository, timezone, and scheduler/worker tuning — are supplied through Snowflake Application Configuration, not by editing any deployment files.

5.2 Privileges & Network Configuration

5.2.1 Privileges & Application Roles

Requested account privileges (approved once by you during guided activation):

Privilege	Purpose
CREATE COMPUTE POOL	Run the go-core and Postgres services on a dedicated, app-owned compute pool (MAESTRO_PI_POOL).
BIND SERVICE ENDPOINT	Expose the Maestro-π web UI / REST API as a public ingress endpoint.
CREATE WAREHOUSE	Dedicated warehouse (MAESTRO_PI_WH) for Maestro-π's own Snowflake queries (e.g., the admin user-picker).
IMPORTED PRIVILEGES ON SNOWFLAKE DB	Read SNOWFLAKE.ACCOUNT_USAGE.USERS for the optional admin user-picker screen.

Two application roles control access to Maestro - π: Maestro_pi_user (required to open the app) and Maestro_Pi_admin (which can additionally view and set configuration, and run service-management commands).

Your orchestration data — DAG history, run results, connections, and more — persists across app upgrades; only uninstalling the app removes it.

5.2.2 Network Rules & External Access

Name	Type	Description
MAESTRO_PI_EGRESS_RULE (example name — consumer-chosen)	Network Rule (MODE=EGRESS, TYPE=HOST_PORT), created and owned by the consumer	Allows the GitHub hosts (github.com, api.github.com, codeload.github.com, objects.githubusercontent.com, raw.githubusercontent.com), hooks.slack.com:443, and *.snowflakecomputing.com:443. SPCS only permits well-known ports (22/80/443/1024+); SMTP submission ports (587/465) are rejected, so no SMTP host belongs in this rule.
MAESTRO_PI_EAI (example name — consumer-chosen)	External Access Integration, created and owned by the consumer	ALLOWED_NETWORK_RULES = (the rule above); bound to the app via the Maestro_Pi_eai_ref reference (core.register_eai_callback) — not created or owned by the app itself.

Note: SPCS external access only permits well-known ports (22, 80, 443, and 1024+) — submission ports such as 587/465 are rejected, so only :443 hosts belong in this rule; Gmail SMTP (587) cannot be reached this way. Email alerting requires an HTTPS(:443) provider API or an SMTP relay on a port SPCS allows.

5.3 Connection & Snowflake Privilege Requirements

Maestro - π's application roles (see Sections 3.2 and 4.4) control what a user can do within Maestro - π — manage connections, configure DAGs, trigger/clear runs, and view results. They do not grant privileges on any downstream system a task connects to.

- The application's internal state store does not require any Snowflake privileges of its own — it runs entirely inside your compute pool and needs no additional access from you.
- When a DAG task connects to Snowflake (for example, via the Snowflake operator or a SQL task), the Connection configured for that task must carry a Snowflake role with, at minimum: USAGE on the target warehouse; USAGE on the target database and schema; and the relevant object privileges (e.g., SELECT/INSERT) on the target tables or views.

6. Troubleshooting

Issue	Description	Possible Causes	Recommended Actions
The app doesn't open, or services never get created after install	Nothing loads when you open the app, or the compute pool/services stay absent.	The requested account privileges haven't been approved yet, or your External Access Integration isn't bound.	Approve the requested privileges and bind your EAI in Snowsight, then check CALL CORE.GET_SERVICE_STATUS();. If it's still pending after a few minutes, run CALL CORE.TRY_CREATE_SERVICE(); to retry.
DAGs don't appear after a Git push	A new or changed DAG isn't showing up in the DAGs list.	The Git repository/branch/subdirectory settings aren't set correctly, or the change hasn't synced yet.	Confirm the Git settings under Configurations, then trigger a manual sync from the UI (or wait for the next automatic sync). Check the service logs for clone errors if it still doesn't appear.
A configuration change had no effect	You changed a setting in the Configurations tab, but nothing changed.	Configuration values only take effect after the service is refreshed.	Run CALL CORE.REFRESH_SERVICES(); after any configuration change, then wait about 30–60 seconds and reload the page.
A task or run appears stuck	A task stays in a running state for longer than expected.	The worker that was executing it may have failed or restarted unexpectedly.	Maestro - π automatically detects and recovers stuck tasks (retrying or failing them) within a couple of minutes — usually no action is needed. If it persists, check the task's logs from the Task Detail panel.
A Snowflake/SQL/HTTP task fails immediately	A task using a remote connector errors on its very first attempt.	The Connection's credentials are missing or incorrect, or the target host isn't reachable.	Open the Connection and re-run Test Connection; confirm your External Access Integration allows the target host if it's outside Snowflake.
Per-DAG access looks different after a Git push	A user's access to a specific DAG changed unexpectedly.	Access rules declared inside the DAG file are re-applied on every sync and can override earlier settings; access you set directly in the Admin UI is always preserved.	Re-apply the intended access from Admin → Roles & Permissions, or adjust the access rules declared in the DAG file.

7. Glossary

Term	Definition
DAG	Directed Acyclic Graph — a pipeline definition (Python file) describing tasks and their dependencies.
Task Instance	One execution attempt of a task within a specific DAG run (dag_id, task_id, run_id, map_index).
XCom	Cross-Communication — the mechanism tasks use to pass small values to one another.
Trigger Rule	Per-task condition (one of 12) determining when it becomes runnable based on upstream state.
Leader / Worker	The single elected scheduling node (Leader) vs. any number of task-executing-only nodes (Worker).
Scheduler Lock	The single-row Postgres table (scheduler_lock) used for leader election via heartbeat.
Zombie Detector	Reliability subsystem that recovers running tasks whose worker died (stale/absent heartbeat).
Triggerer	Component that polls deferred operators/sensors (trigger_instance) and fires them when their condition is met.
Dead Letter Queue (DLQ)	dead_letter_task table capturing terminally failed tasks for review/requeue.
Executor Class	Local / Remote / Sensor classification driving admission control.
Dataset (scheduling)	A named URI whose producer-task updates can trigger consumer DAGs (any/all condition).
Partition	A partition_def-scoped run boundary tracked in partition_status.
Compute Pool	Snowflake-managed pool of VM "nodes" backing the SPCS services (MAESTRO_PI_POOL).
DAG Version	Frozen (tasks, dependencies) snapshot (dag_version, keyed by a SHA256 hash) that pins in-flight runs against DAG edits.
RBAC / per-DAG permission	Application-role grants (Maestro_Pi_permission) plus the per-DAG ACL table (dag_permission) — Maestro - π's analogue of role-based + resource-level access control.
SPCS	Snowpark Container Services — Snowflake's managed platform for running containerized services.
EAI	External Access Integration — the Snowflake object enabling secure outbound network connectivity for a service.
Application Package	The provider-owned container (manifest.yml, setup script, service specs, images) that is versioned and listed on the Snowflake Marketplace; a consumer installs an Application from it.
Application Role	A role scoped to an installed app (e.g., Maestro_pi_user, Maestro_Pi_admin) that the consumer grants to their own Snowflake roles/users to control who can reach the endpoint or administer the app.
Reference (Native Apps)	A manifest-declared slot (e.g., Maestro_Pi_eai_ref) that a consumer binds to one of their own account objects — here, a consumer-owned External Access Integration — at activation time.

8. Reference Artifacts & Editable Links

Asset Type	Description	Official Documentation Link
Snowflake Platform Documentation	SQL, architecture, security, administration, advanced features.	Snowflake Documentation Home
Architecture & Core Concepts	Snowflake architecture, storage, compute, platform concepts.	Snowflake Key Concepts and Architecture
Snowflake Container Services (SPCS)	Overview and concepts for running containerized workloads in Snowflake.	SPCS Overview
Snowflake Services	Creating, managing, and operating Snowflake services.	SPCS Services Guide
Compute Pools	Configuring and managing compute pools for container execution.	Compute pools Guide
Native Application	Deploy Applications on Snowpark container Service	Native Application Guide

9. Appendix

9.1 Environment Deployment Guide

9.1.1 Prerequisites

- Snowflake account that can install Native Apps (Marketplace listings), with SPCS enabled in the target region.
- ACCOUNTADMIN (or an equivalent role) available to approve the app's requested privileges and bind an External Access Integration.
- (Provider-side only) Local Docker environment to build/push the go-core and postgres images into the application package's image repository before publishing a version — not needed by the installing consumer.
- A reachable Git repository (HTTPS) containing DAGs under a configured subdirectory, reachable once the consumer-owned EAI is bound (private repos need credentials available to the container).
- No local secrets file is required — boot-critical secrets (Postgres password, JWT secret, AES key) are generated once, automatically, inside the installed app.

9.1.2 Deployment Steps

10. Create a network rule + External Access Integration you own (Section 5.2.2), covering the GitHub, Slack, and *.snowflakecomputing.com hosts.
11. Click Get App on the Snowflake listing (or, for internal testing, run a dev-mode CREATE APPLICATION ... FROM APPLICATION PACKAGE directly from the package stage) and install it, e.g. as MAESTRO_PI_APP.
12. Open the installed application in Snowsight and complete guided activation: approve the requested privileges (CREATE COMPUTE POOL, CREATE WAREHOUSE, BIND SERVICE ENDPOINT, IMPORTED PRIVILEGES ON SNOWFLAKE DB) and bind your EAI to the Maestro_Pi_eai_ref reference.
13. Wait for guided activation's automatic provisioning — grant_callback and register_eai_callback trigger core.try_create_service(), which creates the compute pool, warehouse, Postgres service, and go-core service (a few minutes on first activation).
14. Open the app (Snowsight Open button, or SHOW ENDPOINTS IN SERVICE ...PI_FLOW_CONTAINER_SERVICE) — the installer is auto-provisioned as the first Admin — then set git_repo_url (and any other) Application Configuration values and apply with CALL CORE.REFRESH_SERVICES();.

9.1.3 Environment Configuration

Boot-critical settings (Postgres connectivity, secrets, bootstrap admin) are generated and wired automatically at install — the consumer never sets them. Consumer-tunable settings are exposed as Snowflake Application Configuration values (git_repo_url, git_branch, git_dags_subdir, SMTP

host/port/user/password/from/TLS, ui_timezone, default_timezone, scheduler_poll_interval_secs, ingestion_poll_interval_secs, worker_min, worker_max) — set via the app's Configurations tab in Snowsight or ALTER APPLICATION ... SET CONFIGURATION VALUE, validated synchronously by core.validate_config, and applied only after CALL CORE.REFRESH_SERVICES();. The only value required to start syncing DAGs is git_repo_url — the app launches successfully without it and simply idles ingestion until it is set.

9.1.4 Validation

- Confirm both requested privileges are approved and the EAI is bound: SHOW GRANTS TO APPLICATION MAESTRO_PI_APP; and check the Maestro_Pi_eai_ref reference on the app's Snowsight page.
- Confirm PI_FLOW_CONTAINER_SERVICE and PI_FLOW_POSTGRES_SERVICE show RUNNING via CALL MAESTRO_PI_APP.CORE.GET_SERVICE_STATUS();.
- Hit the public ingress endpoint (SHOW ENDPOINTS IN SERVICE MAESTRO_PI_APP.SERVICES.PI_FLOW_CONTAINER_SERVICE;) and confirm the installer lands as the first Admin.
- Confirm exactly one row is held in metadata.scheduler_lock with a fresh heartbeat_at (via CALL MAESTRO_PI_APP.CORE.GET_SERVICE_LOGS('0', 'go-core'); or an internal check).
- Set git_repo_url, apply with CALL MAESTRO_PI_APP.CORE.REFRESH_SERVICES();, push a test DAG, and confirm it appears after the next ingestion poll (default 10s) or via a manual sync.

9.2 Maestro - π Setup Guide

Step 1: Create an External Access Integration you own :

Create a network rule + External Access Integration covering the GitHub, Slack, and Snowflake hosts in Section 5.2.2, per Section 9.1.2 step 1 — you own this EAI and bind it to the app during activation.

Step 2: Install the app and approve privileges :

Install from the listing, approve the requested account privileges, and bind your EAI to the Maestro_Pi_eai_ref reference — this triggers automatic provisioning of the compute pool, warehouse, and both services.

Step 3: Open the Maestro - π Web UI

Once both services show RUNNING , click Open on the Snowsight app page (or use SHOW ENDPOINTS IN SERVICE ...PI_FLOW_CONTAINER_SERVICE) to reach the web UI/API on its public ingress endpoint. The installer is auto-provisioned as the first Admin — first-time login and initial setup steps are covered in the End-to-End Demo Walkthrough (Section 9.3).

9.3 End-to-End Demo Walkthrough

9.3.1 Create a Connection

1. Open Connections from the left navigation.

2. Click New Connection and choose a type — Snowflake, Postgres, MySQL, Redshift, S3, GCS, Databricks, HTTP, SSH, or Slack.
3. Fill in the form (fields adapt to the connection type — host, port, credentials, etc.).
4. Click Test Connection to run a live check (e.g. SELECT 1, a bucket listing) before saving.
5. Click Save. Secrets are encrypted at rest (AES-256-GCM) — nothing is stored in plain text.

9.3.2 Add Users and Assign Roles

1. Go to Administration → Users.
2. Click Add User and select the person's Snowflake login from the dropdown (or type it in directly).
3. Tick the role(s) to assign — Admin, Op, Editor, Viewer, or a custom role — then click Create.
4. The user can sign in immediately using their Snowflake identity — no separate password to set.
5. Roles can be adjusted anytime by clicking the role chips next to a user in the Users table.
6. Need something more specific? Go to Roles & Permissions → Create Role, then tick exactly the permissions that role should have.

9.3.3 Author/Apply a DAG

1. Write a DAG as a Python file and place it in the dags/ folder of the connected Git repository.
2. Define the schedule, tasks, and dependencies (>> / <<) in the file.
3. Push the change to the configured branch.
4. PI-Flow picks it up automatically on its next sync cycle — or trigger it immediately via Manual Sync in the UI.
5. Once parsed, the DAG appears in the DAGs list, ready to run.

9.3.4 Trigger a Run

1. Open the DAG from the DAGs list.
2. Click Trigger Run.
3. Fill in the params form if the DAG defines run parameters — values are validated before the run starts.
4. Click Trigger to launch it immediately, or use Backfill to run it across a historical date range.
5. The new run appears instantly, and its tasks begin executing according to the DAG's dependency graph — no further action needed.

9.3.5 Review Results and Failures

1. Open the run from Runs, or use the Grid view to see all runs and tasks briefly.
2. Task states update live on screen as they happen — no manual refresh required.
3. Click any task to view its logs, execution history, and metadata.

4. If a task fails:
- Check whether it's retrying automatically or has moved to the Dead-Letter Queue.
- Read the logs to see what went wrong.
- Use Clear to re-run it (and everything downstream), or Mark Success/Failed to override its state manually.
5. Add a task note if you want to leave context for the next person reviewing it.

10. Support & Contact Information

10.1 Technical Support

For any issues, questions, or bug reports related to Maestro-π, please reach out to the Pi-Snow Support Team using the contact information below :

• Email: Admin@Pibythree.com, Alliance@Pibythree.com

10.2 Business & Sales Inquiries

For questions related to the broader platform roadmap or licensing:

- Website: www.pibythree.com
- Contact : Alliance@pibythree.com

10.3 Documentation & Feedback

To access the latest version of this document or to provide feedback on the application's performance:

• Documentation Portal: Pi Accelerators by PibyThree.