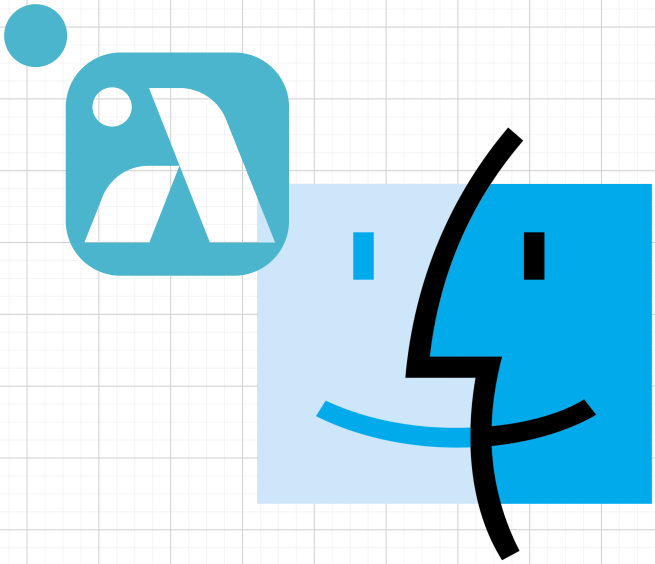


Mac™ OS

Almaden macOS Agent

Gabriel Henrique Hoppe Coelho

Index

About the agent

The Agent is a native macOS application built using **Xcode** and designed to run continuously in the background. Once installed, it resides in the system-wide `/Applications` folder.

To ensure persistent execution, the Agent is managed by a **Launch Agent (or Launch Daemon)**, which starts the app with **root privileges** and automatically restarts it if it crashes or is terminated for any reason. This guarantees a resilient and self-recovering operation.

The Agent is **completely headless**:

- No windows
- No Dock or menu bar icon
- No graphical interruptions to the user experience

To assist with debugging, diagnostics, and internal validation, a dedicated **Command Line Interface (CLI)** will be provided. This CLI tool will offer utilities aimed at **Quality Assurance teams and IT staff**, enabling safe inspection of Agent state, data, and logs when needed.

This architecture ensures maximum transparency to the end user while giving technical teams full visibility and control when required.

Technologies
Swift 5.0
SQLite

Packages	
SQLite.swift	https://github.com/stephencelis/SQLite.swift

Application data

The `AppData` class is a **singleton** responsible for managing the application's local data directory and ensuring required files are created and accessible at runtime. It defines a consistent interface for locating and provisioning the files necessary for the agent to function correctly.

Class Definition

```
class AppData {
    public var appdata_path: URL
    public func setup() throws
}
```

- **appdata_path: URL**
  The base directory used to store application data.
  RELEASE builds: `/Library/Application Support/Almaden/` (SYSTEM)
  DEBUG builds: `~/Library/Application Support/Almaden/` (USER)

- **setup() throws**
  Initializes the AppData object and ensures all required files are created

Required Files

The list of required files is defined in a global constant:

```
let CAppDataFiles = [
    AppDataFile(file_name: "almaden-db.sqlite"),
    AppDataFile(file_name: "consumers-fences.json", initial_content: "{}"),
    AppDataFile(file_name: "settings.json", initial_content: "{}")
]
```

To add a new required file, append a new `AppDataFile` entry to `CAppDataFiles` inside the `Constants` folder.

JSON handler

The `JSONHandler` class is a **singleton** designed to simplify reading and writing key-value pairs in JSON files. It is primarily used to manage configuration or internal state files, such as `settings.json`, within the agent's local data directory.

This handler provides safe, error-aware access to JSON content using straightforward APIs.

Class Definition

```
class JSONHandler {
    func fetch_property(file_url: URL, key: String) throws -> String?
    func set_property(file_url: URL, key: String, value: String) throws
}
```

- **fetch_property(file_url: URL, key: String) throws -> String?**
  Retrieves a value from a JSON file based on the specified key.
  Reads the entire file, parses it as a `[String: String]` dictionary, and returns the value for the key if found.

- **set_property(file_url: URL, key: String, value: String) throws**
  Updates or inserts a key-value pair in the specified JSON file.
  Reads the entire file, modifies the dictionary, and writes the updated content back.

Settings handler

The `SettingsHandler` is a dedicated component responsible for retrieving the agent's configuration values from the `settings.json` file. This file is located inside the application data directory defined by `AppData.shared.appdata_path`.

The `SettingsHandler` is implemented as a **singleton** and acts as a convenient interface over the lower-level `JSONHandler`, ensuring robust and centralized access to configuration data

Class Definition

```
protocol SettingsHandler {
    func fetch_settings() -> ISettings?
}
```

- **fetch_settings() -> ISettings?**
  This is the main public method of the handler. It attempts to read all required configuration fields from `settings.json` and, if successful, returns a populated `ISettings` struct.

  If any required setting is missing or cannot be parsed (e.g., invalid format), the method returns `nil`.

Example `ISettings struct`

```
struct ISettings {
    let customer_email:     String
    let department_name:    String
    let receiver_url:       String
    let receiver_port:      UInt16
}
```

Example `settings.json`

```
{
  "customer_email": "john.doe@example.com",
  "department_name": "IT",
  "receiver_url": "https://receiver.acme.io",
  "receiver_port": "443"
}
```

Polling managers

The `PollingManager` protocol defines a contract for managing recurring tasks in the agent. It provides a unified interface for scheduling, tracking, and controlling background jobs that must execute at defined intervals.

It is used by Readers and other polling entities that gather data periodically.

Class Definition

```
protocol PollingManager {
    var is_alive: Bool { get set }
    var last_execution: Date? { get set }
    var last_succesfully_execution: Date? { get set }
    var polling_job: PollingJobCallback { get set }

    func setup_polling() throws
}
```

- **is_alive: Bool**
  Indicates whether the polling task is currently active and running.
  Useful for preventing duplicate execution or monitoring system activity.
- **last_execution: Date?**
  Timestamp of the most recent execution attempt, whether successful or not.
  Helps in understanding task frequency and detecting failures.
- **last_succesfully_execution: Date?**
  Timestamp of the last successful execution.
  Useful for logic that depends on the last known good state of the polling entity.
- **polling_job: PollingJobCallback**
  The function that performs the actual recurring task.
  Should be safe to run on background threads and handle failure gracefully.
- **setup_polling() throws**
  Initializes the polling mechanism.
  Must be called before starting any polling tasks. Can throw an error if initialization fails.

🧭 Types of Polling Managers

There are two concrete implementations of the `PollingManager` protocol, each designed for different scheduling needs:

1. **CyclePollingManager**

- Designed for high-frequency polling tasks.
- Suitable for entities that perform quick, repetitive checks (e.g., reading CPU usage every 1 second).
- Example use case: `CPUReader`

2. **ScheduledPollingManager**

- Designed for long-interval polling tasks.
- Ideal for jobs where the last execution timestamp is persisted to disk or database.
- Avoids repeated work by checking whether a scheduled interval has passed.
- Example use case: `SysinfoReader` (runs every 8 hours)

SQLite

The `SQLite` class is responsible for managing the agent's local SQLite database. It encapsulates the logic for connecting to the database file and ensuring that all necessary schema migrations are applied during startup.

It is implemented as a **singleton**, meaning only one shared instance exists during the application's lifecycle:

Class Definition

```
class SQLite {
    public func setup() throws
    public func run_migrations(_ connection: Connection) throws
}
```

- **setup() throws**
  Establishes a connection to the SQLite database file.
  `(AppData.shared.appdata_path/almaden-db.sqlite)`

- **run_migrations() throws**
  Executes all registered database migrations

Tables Definition:

You can declare a new table at **`sqlite.database.tables`** file.

```
class SQLiteTables {

    struct ExampleTable {
        static public let _tbl_ =      Expression<Table>("example_table");
        static public let str_col =    Expression<String>("str_col");
        static public let dat_col =    Expression<Date>("dat_col");
    }
    ....
}
```

Samples Table:

```
class SQLiteTables {

    struct CPUConsumptionSamples: SamplesTable {
        static public let _tbl_ =      Expression<Table>("cpu_consumption");
        static public let cpu_usage =  Expression<Double>("cpu_usage");
        static public let timestamp =  Expression<Date>("timestamp");
    }
    ....
}
```

SQLite migrations

Each migration file follows a structured and versioned pattern to ensure clarity and maintainability across database evolution.

Migrations are saved as Swift files named using the format:

`sqlite.database.migrations.<version> (e.g., v1, v2, v3)`

Class Definition

A typical migration file is structured as a versioned class. For example:

```
class SQLiteMigrationsV1 {
    static func example_migration1 {...}
    static func example_migration2 {...}
}
```

- **Migrations:**
  Contains the actual migration operations to be executed, such as creating tables, views, triggers, etc.

Example: Table Definition

```
internal class GpuStats: SamplesTable {

    static public let _tbl_            = Table("gpu_stats")
    static public let model            = Expression<String>("model")
    static public let utilization      = Expression<Double>("utilization")
    static public let memory_used_mb   = Expression<Int>("memory_used_mb")
    static public let timestamp        = Expression<Date>("timestamp")

}
```

Example: Migration Operation

```
static func create_gpu_stats_tbl(_ connection: Connection) throws {

    try connection.run(Tables.GpuStats._tbl_.create(ifNotExists: true) { tbl in

        tbl.column(Tables.GpuStats.model)
        tbl.column(Tables.GpuStats.utilization)
        tbl.column(Tables.GpuStats.memory_used_mb)
        tbl.column(Tables.GpuStats.timestamp)

    })

    try connection.run("PRAGMA gpu_stats = 1")
}
```

SQLite migrations versioning

Table versioning is very simple: when a table is created, it stores its version using the command `PRAGMA <table_name> = 1`. This means that upon creation, the table is considered to be at version one. As the agent evolves, we can query the current version of the table and apply necessary changes accordingly. See the example below:

Example – v1: Initial GPU Table

- **Table:**

```
internal class GpuMetrics {
    static public let _tbl_ =         Table("gpu_metrics")
    static public let gpu_model =     Expression<String>("gpu_model")
    static public let temperature =   Expression<Double>("temperature")
    static public let timestamp =     Expression<Date>("timestamp")
}
```

- **Migration:**

```
static func gpu_metrics_tbl(_ connection: Connection) throws {
    try connection.run(Tables.GpuMetrics._tbl_.create(ifNotExists: true) { tbl in
        tbl.column(Tables.GpuMetrics.gpu_model)
        tbl.column(Tables.GpuMetrics.temperature)
        tbl.column(Tables.GpuMetrics.timestamp)
    })

    try connection.run("PRAGMA gpu_metrics = 1")
}
```

The **`PRAGMA gpu_metrics = 1`** indicates that this table is at **version 1**.

Example – v2: Add New Column `fan_speed`

In a future version (e.g., v2), we may need to extend the schema. To do this:

- **Migration:**

```
static func gpu_metrics_tbl_v2(_ connection: Connection) throws {
    let current_version = try connection.pragmaValue("gpu_metrics") as? Int ?? 0

    guard current_version < 2 else { return }

    try connection.run("ALTER TABLE gpu_metrics ADD COLUMN fan_speed DOUBLE")
    try connection.run("PRAGMA gpu_metrics = 2")
}
```

Now the **`gpu_metrics`** table has the **`fan_speed`** column.

Logger

The **`Logger`** class is a **singleton** responsible for managing logging across the agent. All logs are stored in the agent's local SQLite database, providing persistent and queryable access to runtime events. It supports writing informational, warning, and error logs—either globally or associated with a specific entity (such as a **Reader**, **Collector**, **Consumer**, or **Dispatcher**).

Class Definition

```
class Logger {
    static public func cleanup_entities_logs()
    static public func cleanup_application_logs()
    static public func info(message: String)
    static public func warning(message: String)
    static public func error(message: String)
    static public func info(entity_reference: Entity, message: String)
    static public func warning(entity_reference: Entity, message: String)
    static public func error(entity_reference: Entity, message: String)
}
```

Rotating Logging System

- The logger uses a **rotating mechanism** to manage log size.
- On every new log insertion, it **automatically checks for old entries** and deletes them if necessary.
- This helps prevent unbounded disk usage while keeping recent logs available.

Overloaded Logging for Entities

- Logging methods are **overloaded** to accept an optional `entity_reference`.
- When an entity is provided (e.g., a **`Reader`** or **`Collector`**), the log is stored **alongside the entity's metadata**.
- This enables filtering logs by entity via the CLI or other interfaces.

Samples tables principle

To fully understand how the agent works, it's important to first understand how collected data is stored.

All collected data is saved into **Samples Tables**, which follow a simple convention:
every Samples Table must have a required `timestamp` column of type `Date`.

This design allows us to store both **historical data** and **state-based data** using the same structure.

Historical vs State Data

- **Historical data** represents values that change frequently and are meant to be analyzed over time.
  For example, CPU usage is stored as historical data — we collect multiple samples over time and can later group them by hour or day to analyze trends (e.g., average CPU usage per hour).
- **State data**, on the other hand, represents the current condition of the system at a given point in time.
  Battery health, level, or cycle count are examples of state data — we usually care only about the most recent values, not how they change over time.

Even though they serve different purposes, **both types of data can be stored using the same table model**.
For state data, we may store multiple records over time, but we typically **only query the most recent one** (the record with the latest timestamp).
For historical data, we keep **all samples** and leverage the `timestamp` column to **analyze patterns and trends** over time.

Battery Info (State data)			
level	cycles	health	timestamp
0.5	221	0.8	2025-05-28T15:00
0.4	221	0.8	2025-05-28T14:39
0.38	0.4	0.8	2025-05-28T14:28

Battery Info (State data)			
level	cycles	health	timestamp
0.5	221	0.8	2025-05-28T15:00

We fetch the row which has the highest timestamp field.

CPU Consumption (Historical data)		
usage	frequency	timestamp
0.5	3.4	2025-05-28T15:31
0.2	1.1	2025-05-28T15:10
0.34	2.3	2025-05-28T14:43
0.48	3.2	2025-05-28T14:11
0.21	1.1	2025-05-28T13:39
0.73	3.9	2025-05-28T13:23

CPU Consumption (Historical data)		
usage	frequency	timestamp
0.35	2.25	2025-05-28T15:00
0.41	2.75	2025-05-28T14:00
0.47	2.5	2025-05-28T13:00

We fetch the all rows and group them by hour (AVG %CPU)

Storing Lists in Samples Tables

Let’s consider an example: suppose we want to store a **state-like value**, such as a **list of available Wi-Fi networks**.
In this case, we are only interested in the **latest collected list**, not in its historical evolution.

At first glance, it seems sufficient to simply fetch all rows from the table with the **latest timestamp** — these would represent the most recent list.

However, a problem arises when the list is **empty**. Since the table itself is a list of individual items, if the last collected list contains zero items, **no records will be inserted**, and there's **no way to tell** that an empty list was collected — it appears as if the collector never ran.

The Solution: Collector Activity Tracking

To solve this, we introduce an auxiliary table called **`collectors_activity`**.

Each time a `Collector` inserts data into a Samples Table, it also updates this **`collectors_activity`** table.
This table has three columns:

- `collector_name`: the unique name of the collector
- `last_package_retrieved_at`: the timestamp representing the time of the latest collected package (even if empty)
- `last_package_stored_at`: the timestamp of the last successful data insertion into the Samples Table

With this design, you can now determine:

- If a collector has run, even if **no items** were stored (by looking at `last_package_retrieved_at`)
- What timestamp corresponds to the **latest group of items** inserted (by comparing to the `timestamp` column in the Samples Table)

To retrieve the most recent list collected (including the empty case), simply fetch all rows in the corresponding Samples Table whose `timestamp` matches the collector’s `last_package_retrieved_at`.
If no rows are found, you can still infer that the list was empty based on the activity table.

Collectors activity		
name	lpr	lps
wifi	2025-05-28T10:13:11	2025-05-28T10:13:12
cpu	2025-05-28T07:23:16	2025-05-28T07:23:17
mem	2025-05-28T08:30:49	2025-05-28T08:30:50

- **lpr:**
  last_package_retrieved_at
- **lps:**
  last_package_stored_at

Scenario 1 (Using above collectors activity table as reference)

Wifi List (State data)			
ssid	channel	rssi	timestamp
wiBA	6	-45 dBm	2025-05-28T10:13:11
wiX	6	-38 dBm	2025-05-28T10:13:11
wiY	6	-34 dBm	2025-05-28T09:28:49

→

Wifi List (State data)			
ssid	channel	rssi	timestamp
wi1	6	-45 dBm	2025-05-28T10:13:11
wi2	6	-38 dBm	2025-05-28T10:13:11

Items which timestamp = lpr = 2025-05-28T10:13:11

(Using above collectors activity table as reference)

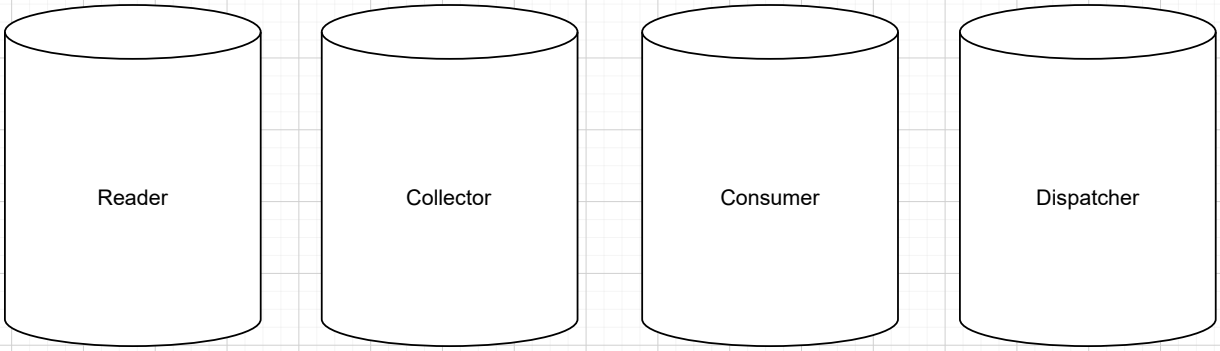
Wifi List (State data)			
ssid	channel	rssi	timestamp
wiY	6	-45 dBm	2025-05-28T09:28:49

→

Wifi List (State data)			
ssid	channel	rssi	timestamp

Items which timestamp = lpr = 2025-05-28T10:13:11

Core agent entities



Reader

The Reader is responsible for accessing raw system information (e.g., memory, CPU, connected devices).

`Readers` do not persist data. Instead, they forward successful readings to the next stage in the pipeline via a callback mechanism. These readings conform to a common **`ReaderData`** structure that includes metadata such as the name, description, and timestamp of the retrieval.

Collector

The Collector is responsible for storing structured data provided by a Reader into an SQLite table.

- Each collected entry is saved with a corresponding timestamp indicating when the read happened.
- In addition, the Collector always updates the **`collector_activity`** table with the timestamp of its last execution, even if no data was saved (e.g., the collected value was empty or irrelevant).

Consumer

The **Consumer** queries the data stored by **Collectors** to prepare information for dispatching.

- It typically performs lookups in SQLite tables populated by Collectors.
- Optionally, the Consumer may also consult the **`collector_activity`** table when necessary — for instance, to determine the most recent collection time.
- This is especially useful when the data type is a list and the Consumer is interested only in the most recent state. Since an empty list cannot be represented as a record in SQLite, the **`collector_activity`** timestamp helps the Consumer understand that a valid collection did occur, even if no data was stored.

This structure enables the Consumer to interpret both the **content** and the **absence** of data meaningfully.

Dispatcher

The **Dispatcher** takes consumed data and sends it to an external service (usually via HTTP or another transport layer).

- Converts the data into the required format (e.g., JSON payload).
- Manages retries and error handling for failed transmissions.
- Upon successful dispatch (e.g., HTTP 200), it instructs the Consumer to update the fence and prune data if needed.

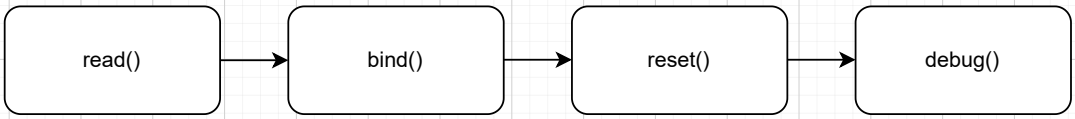
Reader definition

```
protocol Reader: Entity {

    var current_data: (any ReaderData)? { get set }

    func setup() throws
    func bind()
    func reset()
    func read()
    func debug()
}
```

Reader d**efinition** explanation

- **current_data**:
  Holds the most recent successfully bound `ReaderData`. It represents the latest valid snapshot of the system information retrieved.
- **setup()**:
  Initializes the Reader. This may involve allocating resources, subscribing to notifications, or preparing internal state. It is called once before the reader begins operation.
- **bind()**:
  Validates and groups the collected properties into a concrete `ReaderData` structure, only if all required values are non-nil. The result is stored in `current_data`.
- **reset()**:
  Clears all temporary or intermediate properties used during the current read cycle, preparing the reader for a fresh collection.
- **read()**:
  Attempts to collect system data. It is the core operation triggered by the `PollingManager` and is responsible for filling temporary properties with new values.
- **debug()**:
  Logs the current internal state of the reader (e.g., raw values read from the system). Used mainly in development or diagnostic builds.

Reader execution flow



1. **read()**
   Attempts to read system information. Each field (e.g., CPU load, RAM usage, device count) is read individually. Some fields may fail and remain nil.
2. bind()
   If all required values are valid (non-nil), a `ReaderData` instance is created (e.g., `IMemoryConsumptionRD`) and stored in **`current_data`**.
3. **reset()**
   Clears temporary variables to ensure no stale values remain before the next cycle.
4. **debug()**
   If the build is in DEBUG mode, this method outputs diagnostic information such as the raw values read.

Collector definition

```
protocol Collector: Entity {

    var last_inserted_data: (any ReaderData)? { get set }

    func collect(_ reader_data: (any ReaderData)?)
    func debug()

}
```

Collector definition explanation

- **last_inserted_data**:
- Stores the most recent `ReaderData` that was successfully persisted to the database. It is used to avoid redundant writes — if the Reader's latest data has the same timestamp as `last_inserted_data`, the Collector knows the data hasn't changed and skips writing.
- **collect()**:
  The main method responsible for verifying if the ReaderData is a fresh data, and if so, persisting it into an appropriate SQLite table.
- **debug()**:
  Outputs internal diagnostic information related to the Collector's state — useful for debugging data persistence logic, such as seeing what was saved or skipped.

Consumer definition

```
protocol Consumer: Entity {

    var target_samples_table: (any SamplesTable.Type) { get }

    func retrieve() -> ConsumerData?
    func get_current_fence() -> Date
    func set_current_fence(data: ConsumerData)
    func prune_before_fence()
}
```

Consumer definition explanation

- **retrieve()**
  Responsible for querying data previously stored by a `Collector`. This method typically filters data based on the current fence (a timestamp) and prepares it for delivery. The returned `ConsumerData` should include both the payload and a reference timestamp indicating how far the data extends.
- **get_current_fence()**
  Returns the timestamp (fence) up to which data has already been processed or dispatched. This value is used to avoid processing or sending the same data multiple times.
- **set_current_fence(data:)**
  Called after a successful dispatch operation. This method records the new fence value (typically the latest timestamp found in the data just sent), ensuring that future calls to `retrieve()` only consider newer data.
- **prune_before_fence()**
  Deletes or archives data that is older than the most recently confirmed fence. This is useful for controlling storage usage and ensuring the database only contains relevant, unprocessed information.

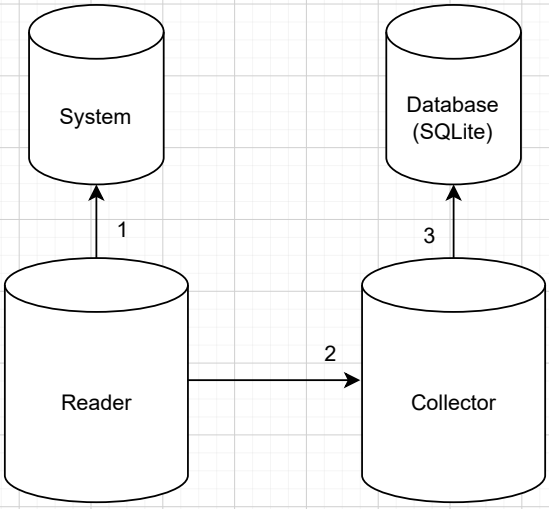
Dispatcher definition

```
protocol Dispatcher: Entity {

    func dispatch(_ consumer_data: any ConsumerData) throws

}
```

Dispatcher definition explanation

- **dispatch()**
  The main dispatch operation, sends data to an external service.
  Handles responses, manages retries, etc...

Read threads



READ THREAD

ReadThread

The **`ReadThread`** is responsible for managing the execution cycle of a reading entity (**`Reader`**) within the agent. It periodically triggers the data collection flow based on the configured polling type and invokes the corresponding **`Collector`** to store the collected data.

Main responsibilities:

- Manage the lifecycle of the reading process: initialization, periodic execution, and interruption.
- Execute the full reading flow of the `Reader`, which includes: `read()`, `bind()`, `reset()`, and `debug()`.
- Call the `collect()` method on the `Collector`, passing the data held in `reader.current_data`.
- Ensure that failures during reading or collecting do not interrupt the thread's execution.

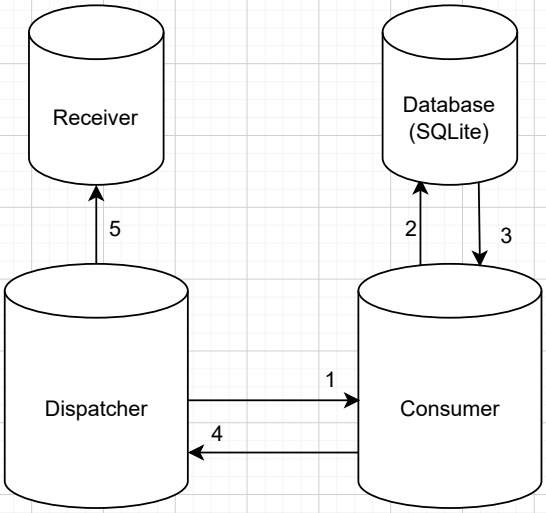
Supported execution types:

- **Cyclic**:
  Uses a **`CyclePollingManager`** to execute at fixed time intervals (e.g., every 1 second).
- **Scheduled**:
  Uses a **`ScheduledPollingManager`** that relies on stored timestamps to determine the next execution time.

Important properties:

- **`reader`:**
  The `Reader` instance responsible for gathering data.
- **`collector`:**
  The `Collector` instance responsible for persisting the data.
- **`polling_type`:**
  Indicates whether the execution is `.cyclic` or `.scheduled`.
- **`interval`:**
  The time interval between executions, used by both polling strategies

Dispatch threads



DISPATCH THREAD

DispatchThread

The `DispatchThread` is responsible for managing the execution cycle of a dispatching entity (**`Dispatcher`**) within the agent. It periodically triggers the `dispatch()` method of the dispatcher, which internally handles the retrieval, transmission, and cleanup of data.

Main responsibilities:

- Manage the lifecycle of the dispatching process: initialization, periodic execution, and interruption.
- Call the `dispatch()` method on the `Dispatcher`, which performs the full dispatching logic internally.
- Ensure that failures during the dispatch process do not stop the execution cycle.

Supported execution types:

- **Cyclic**:
  Uses a **`CyclePollingManager`** to trigger dispatching at fixed time intervals.
- **Scheduled**:
  Uses a **`ScheduledPollingManager`** to decide when to execute next based on stored timestamps.

Important properties:

- **`dispatcher`**:
  The `Dispatcher` instance responsible for the full dispatch operation (retrieval, transmission, and pruning).
- **`polling_type`**:
  Indicates whether the execution is `.cyclic` or `.scheduled`.
- **`interval`**:
  The time interval between executions, used in both polling strategies.

Writing Readers

Readers are responsible for gathering data from the system. Each reader must conform to the `Reader` protocol and implement its core methods: `setup()`, `read()`, `bind()`, and `reset()`.

A typical reader stores intermediate raw data in private properties and uses them to populate the `current_data` property during `bind()`.

You are free to define any custom internal properties your reader needs (e.g., `machine_id`, `host_name`, etc.), as shown below.

Example: `SysinfoReader`

```
final class SysinfoReader: Reader {

    public let type: String
    public let name: String
    public var current_data: (any ReaderData)? = nil

    // Custom internal fields
    private var machine_id: String?
    private var host_name: String?
    private var os_name: String?
    private var os_version: String?

    public init() {
        self.type = CEntityType.READER
        self.name = CReadersNames.SYSINFO_READER_NAME
    }

    // Called once when the agent starts. You can initialize any required resources here.
    public func setup() throws {
        // No setup required for this reader
    }

    // This is where you fetch and assign raw data to internal fields.
    public func read() {
        self.machine_id = self.fetch_machine_id()
        self.host_name = self.fetch_host_name()
        self.os_name = self.fetch_os_name()
        self.os_version = self.fetch_os_version()
    }

    // This method assembles the raw fields into a structured data object (`ReaderData`).
    public func bind() {
        if
            let machine_id = self.machine_id,
            let host_name = self.host_name,
            let os_name = self.os_name,
            let os_version = self.os_version
        {
            self.current_data = ISystemInfoRD(
                __RD__name: "SYSTEM-INFO",
                __RD__description: "General System Information",
                __RD__retrieved_at: Date(),
                machine_id: machine_id,
                host_name: host_name,
                os_name: os_name,
                os_version: os_version
            )
        }
    }

    // Resets the internal state to avoid holding stale data between cycles.
    public func reset() {
        self.machine_id = nil
        self.host_name = nil
        self.os_name = nil
        self.os_version = nil
    }

    // Example fetch methods (mocked)
    private func fetch_machine_id() -> String { "machine-abc123" }
    private func fetch_host_name() -> String { "my-macbook.local" }
    private func fetch_os_name() -> String { "macOS" }
    private func fetch_os_version() -> String { "14.4.1" }
}
```

Writing Collectors

Collectors are responsible for storing the data produced by `Reader` instances. Each collector must conform to the `Collector` protocol and implement the `collect(_:)` method.

Collectors receive data through the `collect(reader_data:)` method — specifically, the `current_data` produced by a `Reader`. It's up to the collector to type-check and persist that data appropriately.

You can define custom logic for deduplication, table schema, and activity tracking per collector.

Example: `SysInfoCollector`

```
final class SysInfoCollector: Collector {

    public let type: String
    public let name: String
    public var last_inserted_data: (any ReaderData)?

    public init() {
        self.type = CEntityType.COLLECTOR
        self.name = CCollectorsNames.SYSINFO_COLLECTOR_NAME
        self.last_inserted_data = nil
    }

    public func collect(_ reader_data: (any ReaderData)?) {

        // Ensure a valid database connection
        guard let database_connection = SQLite.shared.database_connection else {
            Logger.warning(entity_reference: self, message: "No database connection")
            return
        }

        // Ensure the data is of the expected type
        guard let sysinfo = reader_data as? ISystemInfoRD else {
            Logger.warning(entity_reference: self, message: "Reader has no data")
            return
        }

        // Avoid duplicate insertion if data hasn’t changed
        if self.last_inserted_data?.__RD__retrieved_at == sysinfo.__RD__retrieved_at {
            Logger.warning(entity_reference: self, message: "Reader has no updated data")
            return
        }

        do {
            // Use a transaction to insert data atomically
            try database_connection.transaction {

                try database_connection.run(SQLiteTables.SysInfoSamples._tbl_.insert(
                    SQLiteTables.SysInfoSamples.machine_id     <- sysinfo.machine_id,
                    SQLiteTables.SysInfoSamples.host_name      <- sysinfo.host_name,
                    SQLiteTables.SysInfoSamples.os_name        <- sysinfo.os_name,
                    SQLiteTables.SysInfoSamples.os_version     <- sysinfo.os_version,
                    SQLiteTables.SysInfoSamples.timestamp      <- sysinfo.__RD__retrieved_at
                ))

                // Update the collector activity table
                try self.update_collector_activity(
                    connection: database_connection,
                    last_package_retrieved_at: sysinfo.__RD__retrieved_at,
                    last_package_stored_at: Date()
                )
            }

            self.last_inserted_data = sysinfo
            self.debug()

        } catch {
            Logger.error(entity_reference: self, message: "Unexpected error: \(error.localizedDescription)")
        }
    }
}
```

Writing Consumers

Consumers are responsible for **retrieving data** from the samples tables and providing it to the `Dispatcher`. Each consumer must conform to the `Consumer` protocol and implement the `retrieve()` method.

The consumer defines which table it reads from and uses a *fence* mechanism to ensure it retrieves only new data that hasn't been dispatched yet.

Example: `SysinfoConsumer`

```
final class SysinfoConsumer: Consumer {

    public let type: String
    public let name: String
    public var target_samples_table: (any SamplesTable.Type)

    public init() {
        self.type = CEntityType.CONSUMER
        self.name = CConsumersNames.SYSINFO_CONSUMER_NAME
        self.target_samples_table = SQLiteTables.SysInfoSamples.self
    }

    public func retrieve() -> ConsumerData? {

        guard let database_connection = SQLite.shared.database_connection else {
            Logger.warning(
                entity_reference: self,
                message: "Unable to retrieve consumer data: no database connection available."
            )
            return nil
        }

        do {
            let current_fence = self.get_current_fence()

            // Query samples table for newest item after the current fence
            let query = SQLiteTables.SysInfoSamples._tbl_
                .filter(SQLiteTables.SysInfoSamples.timestamp > Expression<Date>(
                    DateHelpers.to_string(target_date: current_fence)
                ))
                .order(SQLiteTables.SysInfoSamples.timestamp.desc)
                .limit(1)

            if let row = try database_connection.pluck(query) {
                return ISystemInfoCD(
                    __CD__fence:    row[SQLiteTables.SysInfoSamples.timestamp],
                    machine_id:     row[SQLiteTables.SysInfoSamples.machine_id],
                    host_name:      row[SQLiteTables.SysInfoSamples.host_name],
                    os_name:        row[SQLiteTables.SysInfoSamples.os_name],
                    os_version:     row[SQLiteTables.SysInfoSamples.os_version]
                )
            } else {
                Logger.warning(
                    entity_reference: self,
                    message: "No consumer data found newer than current fence (\(current_fence))"
                )
                return nil
            }

        } catch {
            Logger.error(
                entity_reference: self,
                message: "Failed to retrieve consumer data | \(error.localizedDescription)"
            )
            return nil
        }
    }
}
```

Writing a Dispatcher

The `Dispatcher` is responsible for sending or forwarding the data collected and processed by the agent. It works by retrieving data from its associated `Consumer`, then dispatching this data to the desired destination, such as a remote server.

In this example, the `SysInfoDispatcher` retrieves system information data from the `SysinfoConsumer`, performs the dispatch operation (currently a placeholder for an HTTP request), and manages the state of the consumer by setting a fence and pruning old data.

```
final class SysInfoDispatcher: Dispatcher {

    public let type: String
    public let name: String

    public init() {
        self.type =     CEntityType.DISPATCHER
        self.name =     CDispatchersNames.SYSINFO_DISPATCHER_NAME
    }

    internal func dispatch(_ consumer_data: any ConsumerData) {
        if let sysinfo_data = consumer_data as? ISystemInfoCD {

            // TODO: Add HTTP request here
            // print("data: \(sysinfo_data)")

        }
    }
}
```

Unit Tests

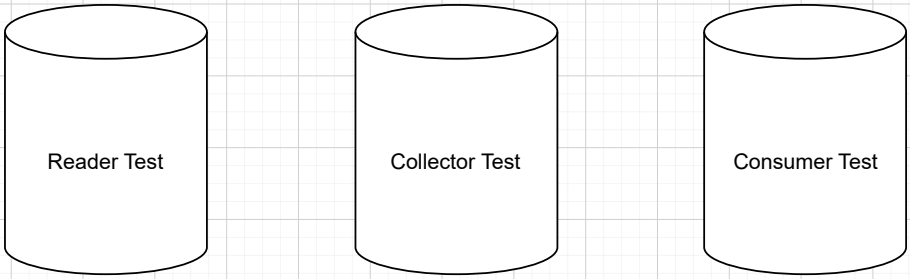
The agent is designed around a modular data pipeline composed of three main types of entities: **Readers**, **Collectors**, and **Consumers**. To ensure correctness and reliability at every stage of this pipeline, the codebase implements **targeted unit tests** for each entity **independently**.

Purpose

The main goal of unit testing in this architecture is to:

- Verify **each component in isolation**.
- Catch regressions early.
- Ensure **data integrity** across the pipeline: from reading → storing → consuming.

Unit tests entities



- **Reader Test:**
  1 - Checks that data is **successfully retrieved** — i.e., not `nil` or empty.

  2 - Validates the **contents of the data**, asserting that fields are **consistent and logically correct**.

- **Collector Test:**
  1 - Ensures that the right **number of rows are written**.

  2 - Asserts that the **column values in the database** match the expected input data.

  3 - Also verifies that a new entry is written to the **`collectors_activity`** table, even when the collected dataset is empty — ensuring that collector activity is always tracked.

- **Consumer Test:**
  1 - Validates that the consumer correctly **retrieves only new data**, starting from the last recorded fence.

  2 - Checks that the **fence is updated**.

  3 - Ensures that **data older than the fence is pruned** correctly, maintaining database hygiene.

Readers unit test

```
import XCTest

/// Example unit test for the `SysinfoReader`,
/// which collects basic system information such as machine ID and OS version.
final class SysinfoReaderTest: ReaderTest {

    /// The reader instance under test.
    public var reader: (any Reader) = SysinfoReader()

    /// Closure that checks the validity of the data retrieved by the reader.
    /// Ensures that all essential fields are populated.
    public var retrieved_data_checker = { (current_data: (any ReaderData)) in

        // Ensure the returned data is of the expected type.
        guard let current_data = current_data as? ISystemInfoRD else {
            XCTFail("Expected 'current_data' to be of type ISystemInfoRD, got \(type(of: current_data))")
            return
        }

        // Validate that all required fields are non-empty.
        XCTAssertNotEqual(current_data.machine_id, "", "'machine_id' should not be empty")
        XCTAssertNotEqual(current_data.host_name, "", "'host_name' should not be empty")
        XCTAssertNotEqual(current_data.os_name, "", "'os_name' should not be empty")
        XCTAssertNotEqual(current_data.os_version, "", "'os_version' should not be empty")
    }
}
```

Collector unit test

```
import XCTest

/// Example unit test for the `SysinfoCollector`,
/// which stores system information retrieved by the `SysinfoReader`.
final class SysinfoCollectorTest: CollectorTest {

    /// The collector instance under test.
    public var collector: (any Collector) = SysinfoCollector()

    /// The table where the collector stores its data.
    public var samples_table: (any SamplesTable.Type) = SQLiteTables.SysInfoSamples.self

    /// A sample `ReaderData` input to be passed to the collector.
    public let reader_data: (any ReaderData) = ISystemInfoRD(
        __RD__name:         "SYSTEM-INFO",
        __RD__description:  "General System Information",
        __RD__retrieved_at: Date(),
        machine_id:         Tools.Generator.random_string(),
        host_name:          Tools.Generator.random_string(),
        os_name:            Tools.Generator.random_string(),
        os_version:         Tools.Generator.random_string()
    )

    /// Function that checks whether the collector wrote the correct rows to the database.
    public func rows_checker(_ rows: [Row]) {

        let records = rows.map { ISysInfoSamplesRecord(row: $0) }

        // Ensure at least one record was written.
        XCTAssertFalse(
            records.isEmpty,
            "Collector did not persist any records in 'sys_info_samples'"
        )

        // There should be exactly one row per 'read' op.
        XCTAssertEqual(records.count, 1, "Collector should persist exactly one record per read operation")

        // Validate that the saved values match the original reader data.
        guard let record = records.first else { return }
        guard let source = self.reader_data as? ISystemInfoRD else { return }

        XCTAssertEqual(record.machine_id,  source.machine_id,  "Mismatch in 'machine_id'")
        XCTAssertEqual(record.host_name,   source.host_name,   "Mismatch in 'host_name'")
        XCTAssertEqual(record.os_name,     source.os_name,     "Mismatch in 'os_name'")
        XCTAssertEqual(record.os_version,  source.os_version,  "Mismatch in 'os_version'")
    }
}
```

Consumer unit test

```
final class SysinfoConsumerTest: ConsumerTest {

    public let consumer: (any Consumer) = SysinfoConsumer()
    public let samples_table: (any SamplesTable.Type) = SQLiteTables.SysInfoSamples.self

    public let provide_data = ISysInfoSamplesRecord(
        machine_id: Tools.Generator.random_string(),
        host_name: Tools.Generator.random_string(),
        os_name: Tools.Generator.random_string(),
        os_version: Tools.Generator.random_string(),
        timestamp: Date()
    )

    public func provide() {
        guard let database_connection = SQLite.shared.database_connection else {
            XCTFail("No database connection")
            return
        }
        do {
            try database_connection.run(SQLiteTables.SysInfoSamples._tbl_.insert(
                SQLiteTables.SysInfoSamples.machine_id <- provide_data.machine_id,
                SQLiteTables.SysInfoSamples.host_name <- provide_data.host_name,
                SQLiteTables.SysInfoSamples.os_name <- provide_data.os_name,
                SQLiteTables.SysInfoSamples.os_version <- provide_data.os_version,
                SQLiteTables.SysInfoSamples.timestamp <- provide_data.timestamp
            ))
        } catch {
            XCTFail("Error while trying to provide data: \(error)")
        }
    }

    public func check_retrieved_data(_ consumer_data: (any ConsumerData)) {
        guard let retrieved_data = consumer_data as? ISystemInfoCD else {
            XCTFail("Expected 'consumer_data' to be of type ISystemInfoCD")
            return
        }
        if !DateHelpers.isEqual(retrieved_data.__CD__fence, provide_data.timestamp, precision: .millisecond) {
            XCTFail("Expected fence to be \(provide_data.timestamp), found \(consumer_data.__CD__fence)")
            return
        }
        XCTAssertEqual(retrieved_data.machine_id, provide_data.machine_id)
        XCTAssertEqual(retrieved_data.host_name, provide_data.host_name)
        XCTAssertEqual(retrieved_data.os_name, provide_data.os_name)
        XCTAssertEqual(retrieved_data.os_version, provide_data.os_version)
    }

    public func expected_fence() -> Date {
        return provide_data.timestamp
    }
}
```

Crash reports

If some crash occurs when the agent is running, all information about the actual state of the agent in the moment it crashed will be stored at:

- **DEBUG:**
  ~/Library/Logs/DiagnosticReports/ **(run as user)**
- **RELEASE:**
  /Library/Logs/DiagnosticReports/ **(run as system)**

Debug symbols settings

PROFILE = DEBUG	
Deployment post processing	NO
Strip debug symbols during copy	NO
Strip linked product	NO

PROFILE = RELEASE	
Deployment post processing	NO
Strip debug symbols during copy	YES
Strip linked product	YES

- **Deployment post processing:**
  Performs additional optimizations and cleanup steps after the build process, such as stripping symbols or thinning binaries, to prepare the app for release.
- **Strip debug symbols during copy:**
  Removes debug symbols when copying build products to reduce app size and protect internal implementation details.
- **Strip linked product:**
  Eliminates unused or debug-related symbols from the final linked binary to minimize size and improve performance.

Optimization settings

PROFILE = DEBUG	
Clang compiler optimizations	None [-O0]
Swift compiler optimizations	None [-Onone]

PROFILE = RELEASE	
Clang compiler optimizations	Fatest Smallest [-Os]
Swift compiler optimizations	Optimize for speed [-O]

- **Clang compiler optimizations:**
  Controls optimization levels for C/C++/Objective-C code, such as inlining or loop unrolling, to improve runtime performance or reduce binary size.
- **Swift compiler optimizations:**
  Specifies optimization settings for Swift code, balancing between build speed and runtime performance (e.g., `-O`, `-Osize`, or `-Onone`).

Symbolicating a crash report

Once you get the generated **.ips** file you will need to see two things:

- **Stack Trace:**
  A list of function calls that were active at the time of the crash. It helps identify where the crash occurred and in what order functions were called.
- **PC register:**
  Holds the address of the instruction that was being executed at the time of the crash. Used with the image base to locate the exact crashing line in the code during symbolication.
- **Main executable image base/load address:**
  The memory address where the app's main executable was loaded. This is needed to symbolicate addresses in the crash report and map them to source code.

Stack trace is located in the **`"threads"`** field of the **`.ips`** **JSON file**.
Each thread contains a list of stack frames showing the function call sequence.

PC (Program Counter) register is found in the **`"exception"`** field.
It indicates the exact instruction address where the crash occurred.

Main executable image base/load address is in the **`"usedImages"`** field, typically as the first item.
This provides the load address of the main executable needed for symbolication.

If the **PC (Program Counter) register** points to an address within the **main executable's virtual memory range** at the time of the **crash**, you can directly symbolicate that address using the main **binary's symbols (.dSYM)**.

If it falls outside that range **(e.g., in a system library or a dynamic framework)**, you'll need to inspect the **stack trace** to identify the crash context.

After identifying the exception address using either the stack trace or the PC register, you can use `atos` to symbolicate it and retrieve the corresponding function or source line:

```
atos -o almaden-agent.app.dSYM/Contents/Resources/DWARF/almaden-agent /
    -arch arm64
    -l 0x100D30000
    -I 0x100D51BC0
```

Explanation of flags:

- `-o` specifies the path to the dSYM file containing debug symbols.
- `-arch` specifies the architecture (e.g., arm64).
- `-l` sets the **load address** (the base address where the binary was loaded in memory).
- `-i` sets the **instruction address** (the PC or address from the stack trace to be symbolicated).

Example output:

```
Agent.entry_point() (in almaden-agent) (Agent.swift:145)
AppDelegate.applicationDidFinishLaunching(_:) (in almaden-agent) (delegate.swift:15)
@objc AppDelegate.applicationDidFinishLaunching(_:) (in almaden-agent) (<compiler-generated>:13)
```

This resolves the call stack, including inline frames, and maps addresses back to their original source code locations.

In this case, the crash occurred in the `Agent.swift` file, at line 145.

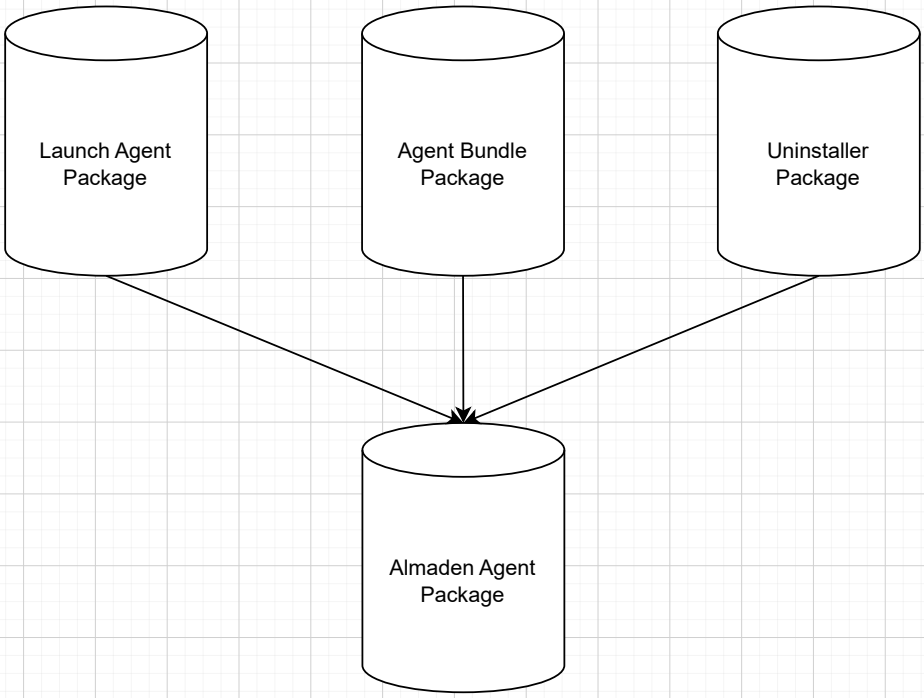
Agent installer

The Almaden macOS agent will be packaged as a **`.pkg`** installer. We chose this format because **`.pkg`** files can be installed **silently** via a terminal command or simply by **double-clicking** the file.

If the customer has a distribution system, they can deploy the **`.pkg`** file along with a shell script that executes the installation command. If not, the user can manually download the **`.pkg`** file and double-click it to install the agent.

We'll build the **`.pkg`** using an Apple-native tool called **`pkgbuild`**. This approach allows us to maintain one repository for the installer and a separate one for the agent's source code.

Agent packages



- **Launch Agent Package:**
  It will be responsible to setup the launchd service (that's what keep our process alive).
- **Agent Bundle Package:**
  It will install the .app bundle at /Applications system folder.
- **Uninstaller Package:**
  This one will create two script files at /Applications/Uninstall folder.

Alma CLI Command Line Interface for Almaden macOS Agent

Alma CLI is a vital tool that provides command-line access to the internal components of the **Almaden macOS Agent**. It allows you to retrieve logs from various entities — such as readers, collectors, consumers, and dispatchers — enabling fast diagnostics and visibility into the agent's behavior.

In addition to log inspection, Alma CLI also supports adjusting key runtime configurations, such as the `receiver_url` and `receiver_port`, giving you flexible control over how and where the agent sends its data.

Lightweight, scriptable, and built for developers and operators alike, Alma CLI is the ideal utility for managing and monitoring the agent in both development and production environments.

Alma CLI commands

`almacli list <type>`

List all entities of the specified type.

Subcommands:

- `reader` – Lists all readers
- `collector` – Lists all collectors
- `consumer` – Lists all consumers
- `dispatcher` – Lists all dispatchers
- `read-threads` - Lists all read threads
- `dispatch-threads` - List all dispatch threads

`almacli entity-logs`

Retrieves logs from specific agent entities (e.g., readers, collectors, consumers, dispatchers).

Options:

- `--level <log-level>`
  `(Optional)` Filter by log level (`info`, `warning`, `error`).
  Defaults to showing all logs.

- `--entity-name <name>`
  `(Optional)` Filters logs by the exact name of the entity.
  If omitted, the command lists all entities along with their log counts.

`almacli application-logs`

Display internal logs from the agent process.

Options:

- `--level <log-level>`
  `(Optional)` Filter by log level (`info`, `warning`, `error`).
  Defaults to showing all logs.

Alma CLI commands

`almacli activity`

Show execution activity of the agent's threads.

Subcommands:

- `read-thread` - Show activity logs for read threads.
- `dispatch-thread` - Show activity logs for dispatch threads.

Options:

- `---name <thread-name>`
  Specify the target thread to retrieve activity information for

`almacli config`

Update or inspect the agent's runtime configuration. This command allows you to dynamically set values like networking parameters, customer metadata, and more — without restarting the agent.

Options:

- `--customer-email <email>`
  Sets the customer's email address associated with the current device where the agent is running.
- `--department-name <name>`
  Sets the department name associated with the current device. (e.g., "IT", "Security").
- `--receiver-url <url>`
  Defines the full receiver URL the agent will send data to (e.g., `http://receiver.mycompany.com`).
- `--receiver-port <port>`
  Defines the port used by the agent to connect to the receiver (e.g., `8080` or `443`).
- `--google-geo-api-key`
  Defines the google geolocation API private key used by the agent.

Usage Notes:

- You can set one or multiple options at once.
- Omitting all options will result in no changes.

`almacli launchd`

Stops and starts the Almaden macOS agent.

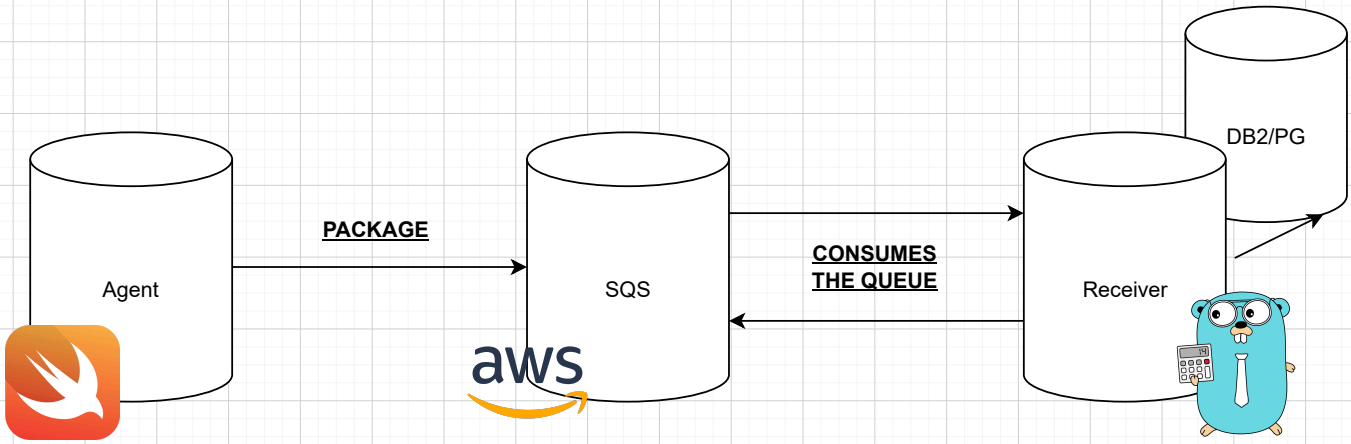
Subcommands:

- `load` - Start the agent
- `unload` - Kill the agent

`almacli device-identifier`

Retrieves the current device ID.

Package Receiver



macOS Agent Flow Summary

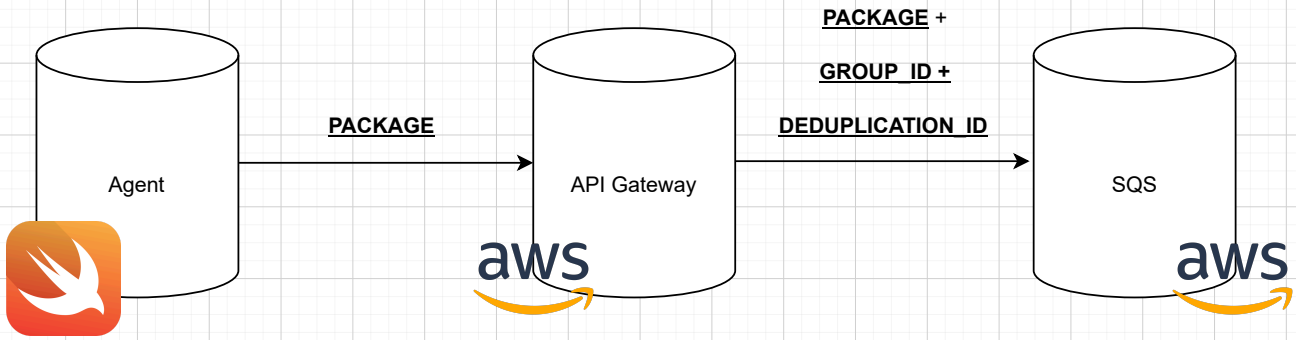
1. **Package submission**
    - The macOS agent collects system information (CPU, Memory, Battery, etc.).
    - Each package is sent to the **AWS API Gateway**, which forwards the request to **AWS SQS**.

2. **Storing in the queue (SQS FIFO Queue)**
    - The package is stored in the queue with two key attributes:
        - **MessageGroupId** → identifies the package type (e.g., `CPU`, `MEMORY`, `BATTERY`).
        - **MessageDeduplicationId** → ensures uniqueness, defined as the **collection timestamp** of the package.

3. **Message consumption**
    - A **Go-based receiver** polls the queue periodically (default: every **10 seconds**).
    - On each request, it fetches up to **10 messages** from the queue.

4. **Processing with worker pools**
    - The receiver dispatches the packages into **worker pools**, one for each package type.
    - Workers process the package and then remove the corresponding message from the queue.

How agent sends packages to the SQS



The macOS agent sends each collected package to the **AWS API Gateway**, which then forwards it to **AWS SQS**.

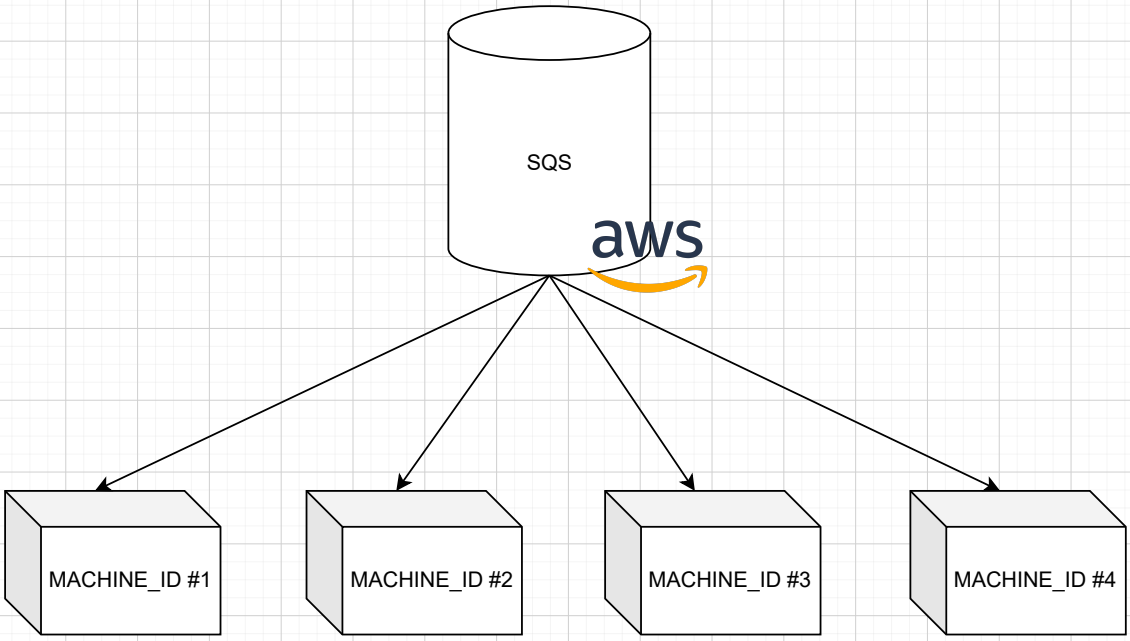
To avoid duplicates, the message is published to the **SQS FIFO queue** using:

- **MessageGroupId** → the **machine-id**.
- **MessageDeduplicationId** → the **package-type + package collection timestamp (ex: WIFI_PACKAGE.2025-04-31T01:00:00)**

This guarantees that:

- Packages are not duplicated in the queue.
- We can easily monitor and track if a duplicate submission attempt occurs.

SQS subqueues

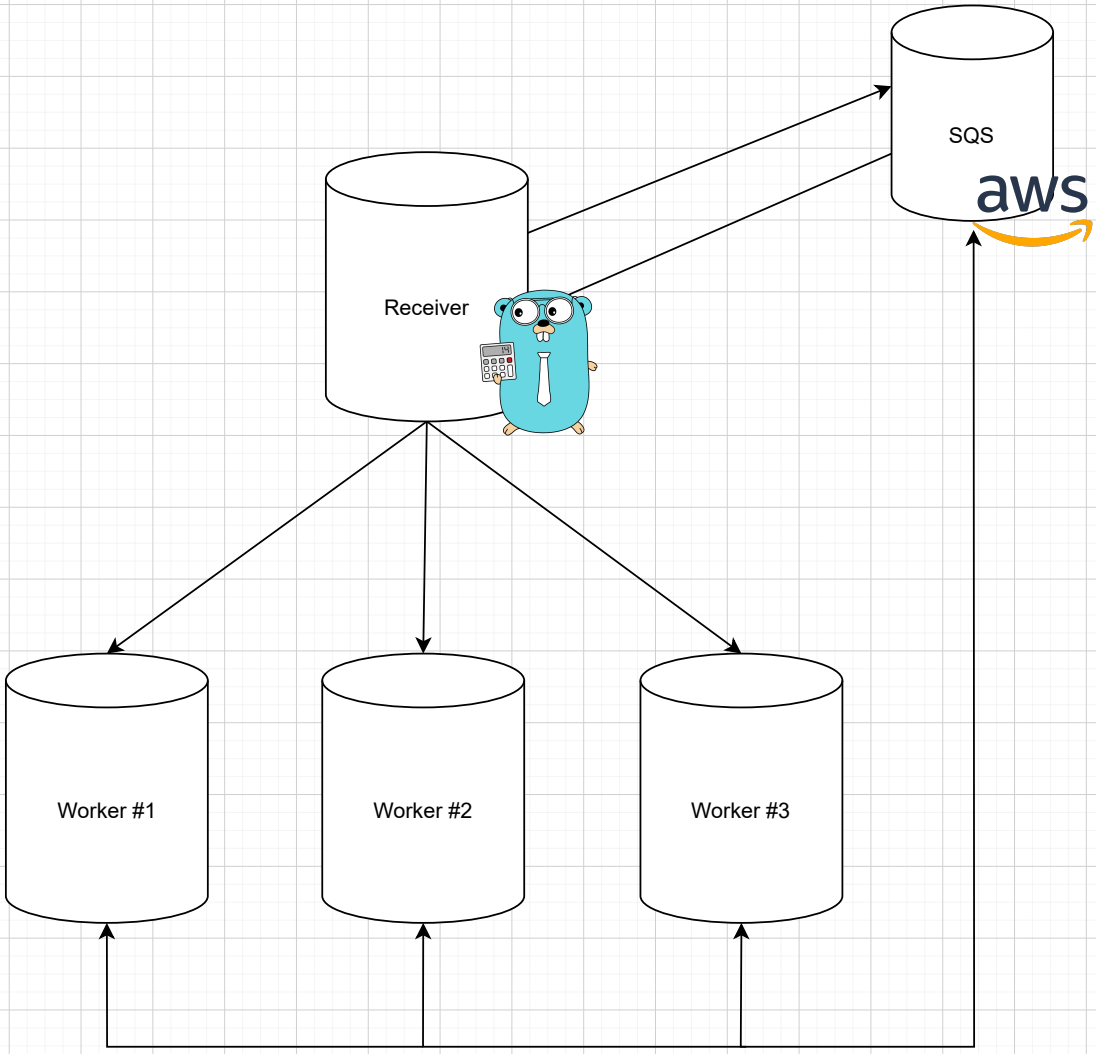


When you request messages from SQS, it will return all messages for a specific **group-id**, as mentioned in the previous page, the **group-id** is always the **machine-id**, so you can think of each device as having its own subqueue inside the SQS service.

In order to receive messages from SQS, you need to specify the **visibility timeout**, which is how long the subqueue will be invisible for other consumers, preventing them from being processed twice.

By default, the macOS receiver is sending 3 minutes for the visibility timeout.

How receiver fetch packages from SQS



When the receiver starts, it creates a set of workers responsible for consuming and handling messages from the SQS Queue.
The number of workers can be configured by updating the **QUEUE_WORKERS** environment variable.