

Run Airflow tasks on your coffee machine

Cedrik Neumann

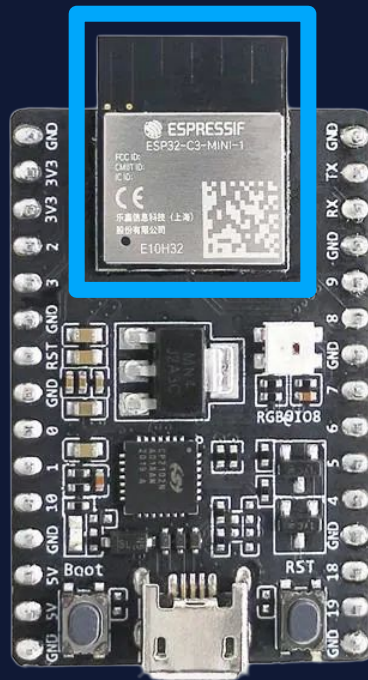
ASTRONOMER

**I don't have a
coffee machine
here today ...**



Meet the Espressif ESP32-C3-MINI-1 SoC

- CPU: single core 32-bit RISC-V
- Frequency: 160 MHz
- Embedded flash: 4 MB
- RAM: 400 KB
- Power: 0.33 W
- You can run Doom on similar hardware*
- **But does it run Airflow?**



* Dual core ESP-32 with 4 MB PSRAM: <https://github.com/espressif/esp32-doom>

Yes, it can run Airflow tasks (Sep 2025)

YES!



- A worker written in Rust
- Dags and tasks written in Rust
- Fully asynchronous
- Compiled to run on bare metal

Why now?

Two major Airflow 3 features made this possible:

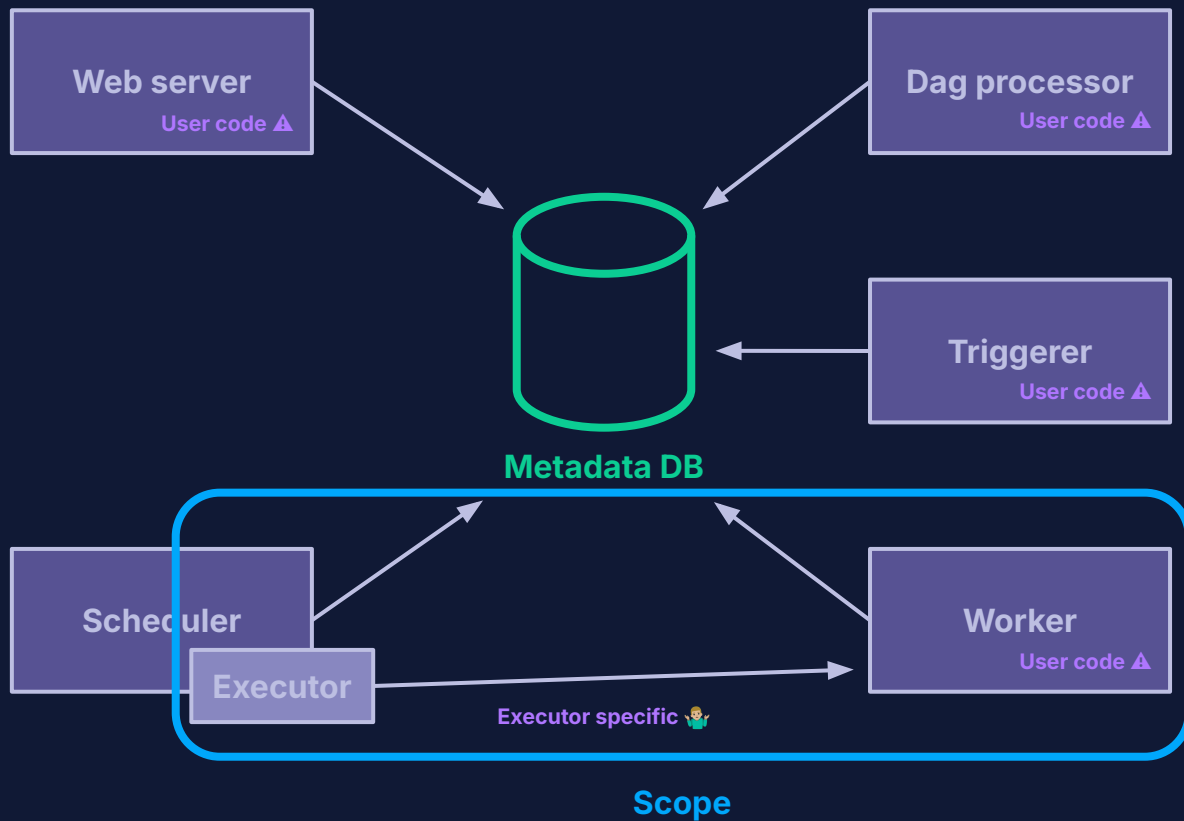
- **Task SDK ([AIP-72](#))**
Interface for the interaction between tasks and Airflow as a HTTP API
- **Edge Executor ([AIP-69](#))**
Pull task execution information from the Executor via a HTTP API

A little bit of background

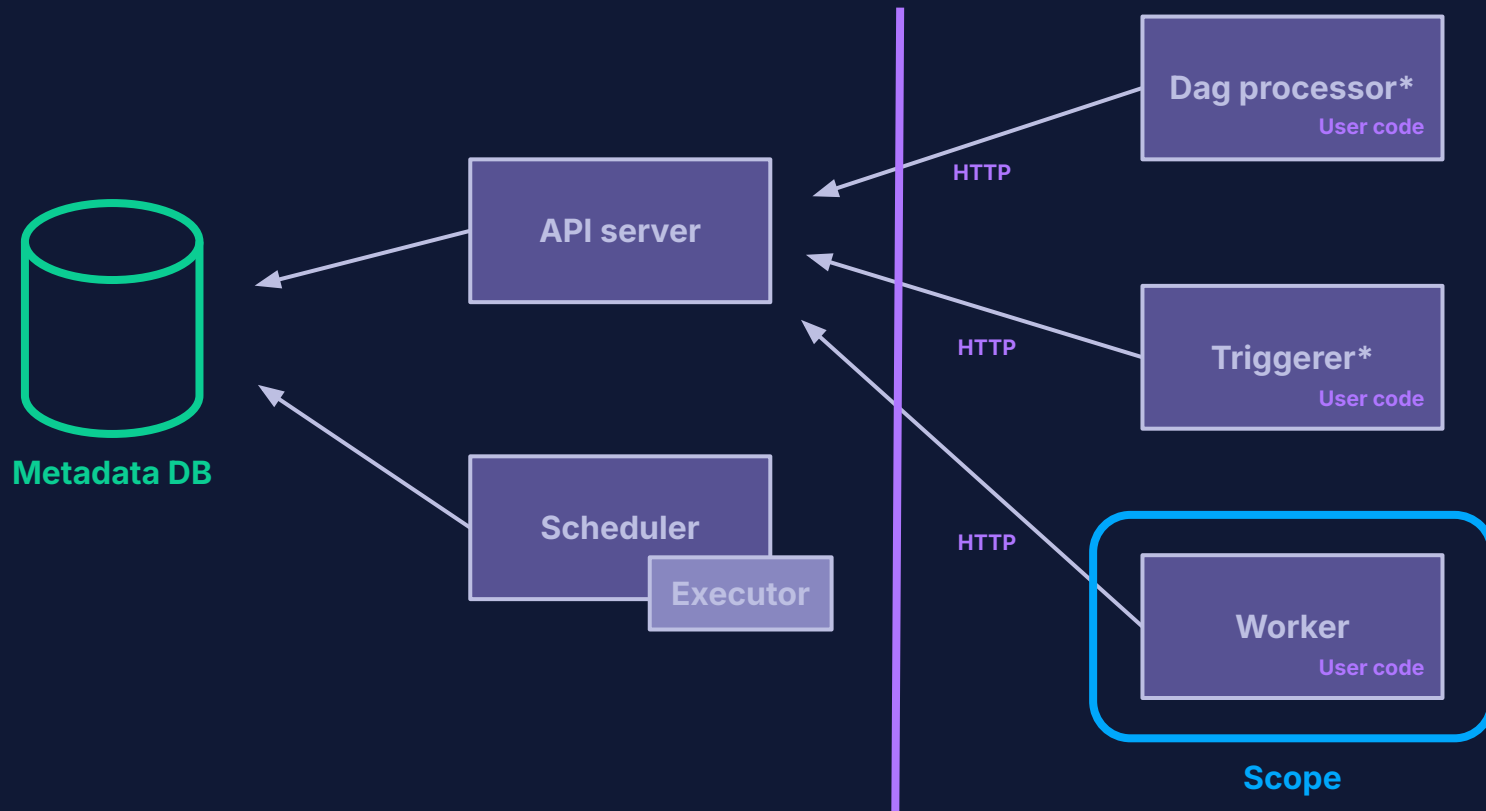
Airflow 2's security model:



Airflow 2 architecture



Airflow 3 w/ edge executor architecture



* In-process API server

Edge Worker Basics

Things you should know

Highlights of the edge API

Worker Register: Initial registration of worker. Fails if version mismatch.

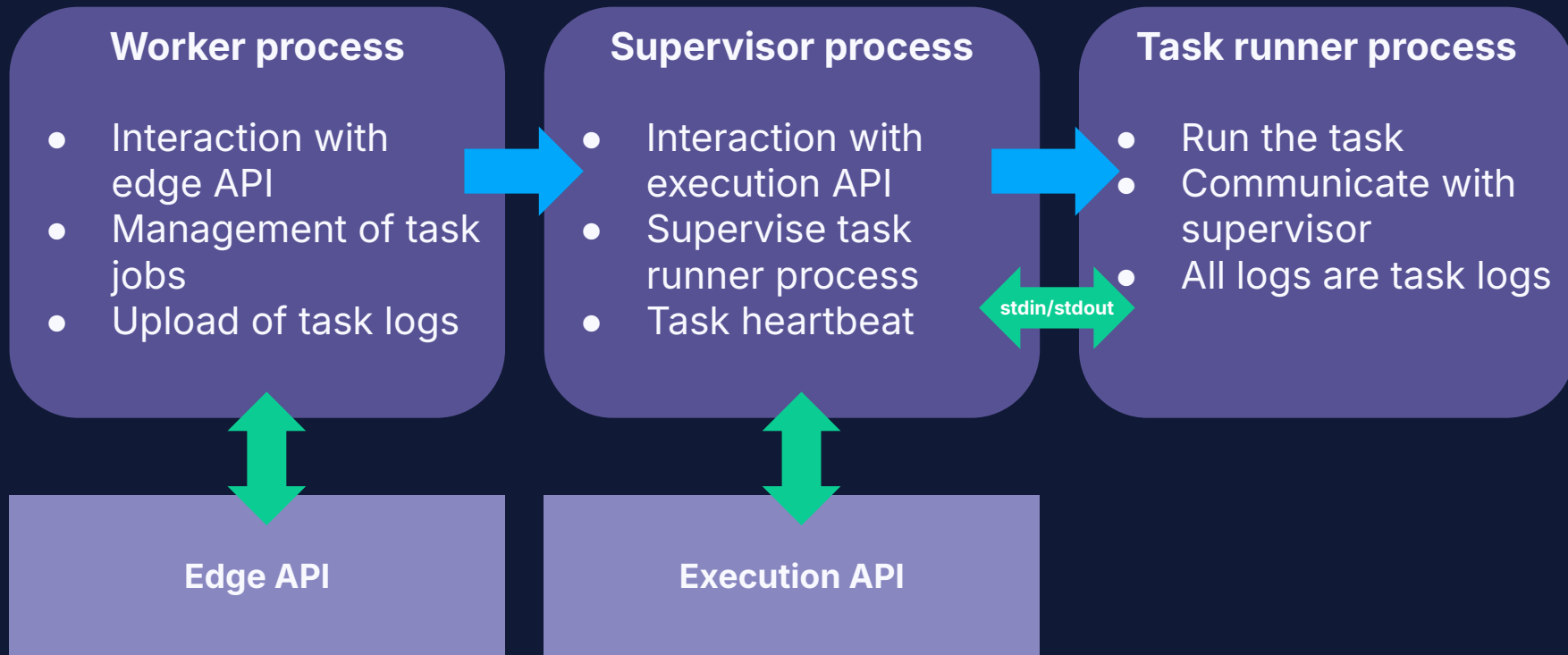
Worker Set State: Used for Heartbeat and state transitions (requested from either side).

Jobs Fetch: Fetch a job to execute on the edge worker.

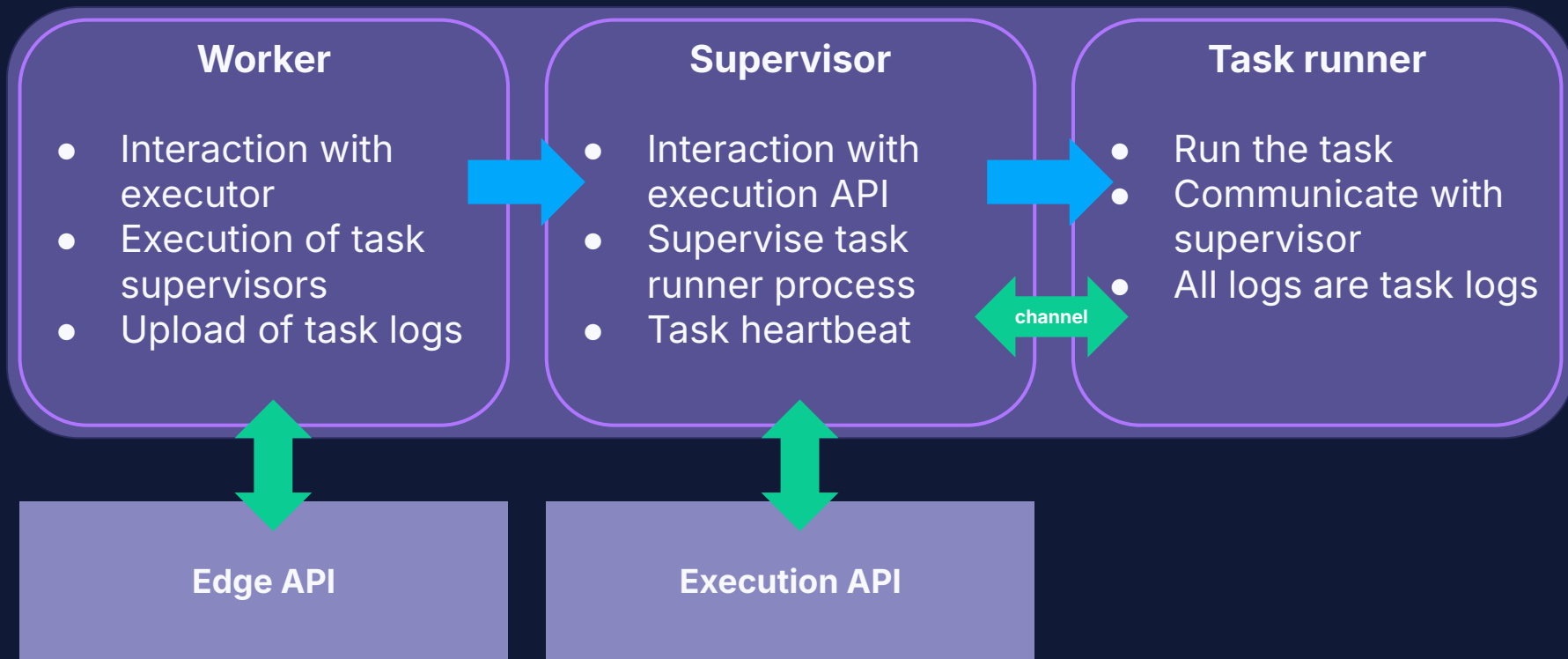
Jobs Set State: Update the state of a job running on the edge worker. Not the same as TI state.

Logs Push: Send logs back to the Airflow deployment. Should be sequential.

Anatomy of an edge worker (Python)



Anatomy of an edge worker (async Rust)



Rust Task SDK

Write Airflow tasks natively in Rust

The Operator trait

- Implement your operator using the `execute` method
- `Output` must be JSON serializable (`serde::Serialize`) or `()`

```
pub trait Operator<R: TaskRuntime> {  
    type Output: JsonSerialize;  
  
    async fn execute<'t>(&'t mut self, ctx: &'t Context<'t, R>) -> Result<Self::Output, TaskError>;  
}
```

Store output in XCom

```
use airflow_task_sdk::prelude::*;
use tracing::info;

#[derive(Debug, Clone, Default)]
pub struct ButtonSensor;

impl<R: TaskRuntime> Operator<R> for ButtonSensor {
    type Output = Button;

    async fn execute<'t>(&'t mut self, _ctx: &'t Context<'t, R>) -> Result<Self::Output, TaskError> {
        info!("Waiting for button press...");
        let button = next_button_pressed().await;
        info!("Button {:?} pressed", button);
        Ok(button)
    }
}
```

Downstream XCom pull

```
[derive(Debug, Clone, Default)]
pub struct LedOperator;

impl<R: TaskRuntime> Operator<R> for LedOperator {
    type Output = ();

    async fn execute<'t>(&'t mut self, ctx: &'t Context<'t, R>) -> Result<Self::Output, TaskError> {
        let button: Button = ctx.task_instance().xcom_pull().task_id("wait_button").one().await?;
        info!("Got button {:?} from upstream task", button);

        // TODO do something with the button

        Ok(())
    }
}
```


Limitations

Not yet implemented:

- Variables & connections
- Full task context
- Assets
- Sensors
- Just build your DagBag and run a worker

This needs some thinking:

- Template rendering
- Dynamic task mapping
- Dag versioning

Python vs Rust



Rust when I have
an atom of
difference between
my type and the
expected type



Python when I
cast a float
into an
unsigned Toyota
Yaris 2023

Learnings from rebuilding Airflow in Rust

Python

- Your task just runs something with an `execute` method
- Convenient globals for access to XCom, Variables, Connections
- Heavy use of inheritance
- Breaking changes often only show up at runtime

Rust

- Ownership impacts how you construct your task perform mutations
- The type system is a safety net while refactoring
- Tight control over what your users can access
- Need to work around the non-existence of inheritance
- Generics can blow up your type definitions (dyn doesn't like async)

Potential use cases

- Resource constrained devices
- Specialized hardware
- Exotic operating systems

Examples:

- Automotive industry
- Household appliances
- Consumer electronics

The Future of the Task SDK?

- **Dag definition interface**

For now a Python Dag must exist in order for it to exist in Airflow. Can we call/execute something which returns back some kind of serialized Dag representation?


- **Plugable supervisor**

Use an existing Python worker to run a non-Python task natively or the other way around.

- **Unified task logs API**

Local paths, remote log storage, edge API ... the way tasks report their logs should not depend on the Airflow setup/configuration.

Demo time!



Home

Dags

Assets

Browse

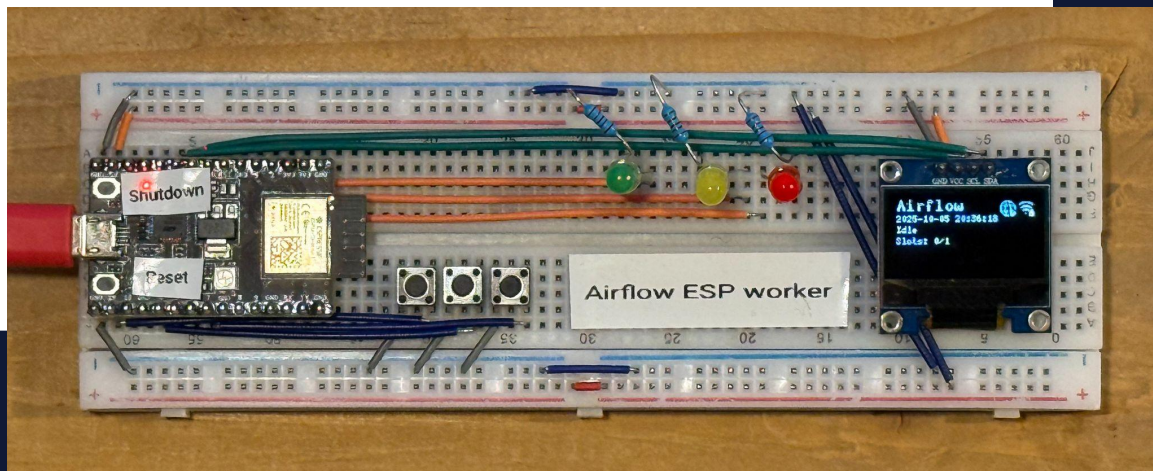
Admin

Security

Edge Worker

Edge Jobs

Worker Name	State	Queues	First Online	Last Heartbeat	Active Jobs	System Information	Operations
airflow-esp	idle	(all queues)	5 days ago	1 second ago	0	<ul style="list-style-type: none">airflow_version: 3.1.0edge_provider_version: 1.3.1concurrency: 1free_concurrency: 1	⇒ ✕ 🔌



Interested in embedded Rust?

Take a look at the [The Rusty Bits](#) YouTube channel.

ESP32 embedded Rust setup explained



Intro to Embassy

Thank you

Airflow Rust SDK

Task SDK and Edge Executor written in Rust

<https://github.com/m1racoli/airflow-rs>

Airflow on ESP

Edge worker running on an ESP32-C3

<https://github.com/m1racoli/airflow-esp/tree/airflow-summit-2025>



**The 2025 Apache
Airflow® Survey**