Prefect — Components Deep Dive

A study guide to Prefect 3 (Python workflow orchestrator), one component at a time: what it is, the code you will type, when to use it, the traps, and where the official docs live.

> **How to read this guide.** Each component has (1) a mental model, usually an analogy, (2) the code, (3) a *when to use / when not to* verdict, (4) traps and (5) 🔗 doc links. The decision tables for **"which way do I run this?"** (§5-6) and **"when is Prefect the wrong tool?"** (§13) are the parts to re-read before choosing an orchestrator. The sibling page *RustFS — Components Deep Dive* covers an S3-compatible store that Prefect can use for results and code.
> **Version note:** written 23 Oct 2026; the docs index lists Prefect 3.9 as the latest OSS line. ⚑ marks details I wrote from prior knowledge rather than re-reading the page this session (decorator parameters, exact import paths, CLI flags). Confirm those against the linked docs and your installed version (`prefect version`) before copying.

---

0. The map: what Prefect is

> **Analogy: an airline operations centre.** Your flights are ordinary Python functions. The operations centre does not fly the plane; it tracks every flight, reschedules on delay (retries), reroutes when an airport closes (infrastructure), logs everything, and alerts when something goes wrong. Planes (your code) fly on your own aircraft (your infrastructure).

- **Workflows are plain Python.** A *flow* is a decorated function; a *task* is a decorated function called inside it. There is no DAG file to declare: dependencies come from data flow and normal control flow (`if`, `for`, `try`), so a workflow can branch and loop dynamically.
- **Orchestration is hybrid.** Your code runs on your infrastructure; the Prefect API (self-hosted server or Prefect Cloud) stores state, schedules, events and automations. Task

runs are orchestrated *client-side* and their updates are logged in batches, so task state in the UI is eventually consistent.
- **Code is not stored by Prefect.** A *deployment* is server-side metadata (which flow, where its code lives, which infrastructure, schedule, parameters). Deployments reference flows defined in Python files; they do not contain code.
- **Prefect 3 changes** (old tutorials differ): workers and work pools replace agents; block-based deployments and storage/infrastructure blocks are the legacy path; assets, transactions and background tasks are first-class. The 2.x to 3.0 upgrade and agents-to-workers guides exist under *how-to-guides/migrate*.

Piece	Where it lives	Job
Flow / task code	Your repo or image	The actual work
Prefect API + database	Cloud or self-hosted server (Postgres)	State, schedules, events, UI
Deployment	API (metadata only)	Run this flow, from here, on that pool, on this schedule
Work pool (+ queues)	API	A channel between deployments and infrastructure
Worker	Your infra (or none for push/managed pools)	Polls the pool and starts runs

Concepts · Quickstart · Upgrade to 3.0 · Migrate from Airflow

---

1. Flows

> **Analogy: a recipe card.** The card lists the steps; any cook with a kitchen can follow it. The kitchen is infrastructure; the order ticket is a flow run.

```
from prefect import flow, task, get_run_logger


@flow(name='daily-etl', retries=1, retry_delay_seconds=60, timeout_seconds=3600, log_prints=True)
def daily_etl(day: str, full_refresh: bool = False) -> int:
    rows = extract(day)
    return load(rows)


if __name__ == '__main__':
    daily_etl('2026-10-22')          # runs locally, recorded in the API if one is configured
```

- A flow is callable like a function. Calling it creates a **flow run** with its own state history. ⚑ Common options: `name`, `retries`, `retry_delay_seconds`, `timeout_seconds`, `log_prints`, `validate_parameters` (type-checks arguments against hints), `persist_result`, `task_runner`, `on_completion` / `on_failure` / `on_cancellation` / `on_crashed` hooks.
- **Subflows:** a flow may call another flow; it becomes a child run with its own state. Flows and tasks can call each other with no required nesting pattern.
- **Flow versus task:** a flow is the unit you *deploy, schedule and parameterise*; a task is a unit you *retry, cache and run concurrently*. Do not put a deployment boundary inside a task.
- **Flow retries rerun the whole flow**; task retries rerun one step. Prefer task retries plus caching so a flow retry does not redo finished work.

Traps. Parameters must be serialisable for deployments (they travel as JSON). Top-level code outside `@flow` runs on every import, including when a worker loads the file. Keep imports light and put heavy setup inside tasks.

📖 Flows · Write and run a workflow · Pass inputs

---

2. Tasks

> **Analogy: one station on the line.** Small, with a clear input and output, and the supervisor logs how long it took and whether it passed.

Tasks are cacheable, retryable units of work with transactional semantics. A task is identified by a task key (a hash of its name and fully qualified function name). The docs recommend **small tasks**: each should be a logical step or a significant side effect, so the UI narrates your workflow.

```
from datetime import timedelta
from prefect import task
from prefect.cache_policies import INPUTS, TASK_SOURCE

@task(retries=3, retry_delay_seconds=[10, 60, 300], timeout_seconds=120,
      cache_policy=INPUTS + TASK_SOURCE, cache_expiration=timedelta(hours=1), log_prints=True)
def extract(url: str) -> dict:
    ...
```

2.1 Three ways to run a task

Call	Blocks?	Returns	Use for
`t(x)`	Yes	The result	Simple sequential steps
`t.submit(x)`	No	`PrefectFuture`, resolve with `.result()` or `.wait()` from the same context	Concurrency inside a flow
`t.delay(x)`	No	Future you need not resolve from the same context	Fire-and-forget background work on separate workers (§10)

- **Dependencies are implicit:** passing an upstream result or future into a downstream task means it cannot start until the upstream is `Completed`. Add `wait_for=[...]` for explicit ordering. `t.map(items)` fans out over an iterable ⚑.
- **Parameter resolution trap (documented):** Prefect walks parameters to resolve nested futures and states. If a dataclass or named tuple field changes, Prefect rebuilds it with `dataclasses.replace()`, which calls `__init__` and `__post_init__` again; Pydantic models use `model_construct()` and skip validators. Use `quote(obj)` to pass a fully resolved object untouched (no future resolution, no dependency tracking), or `opaque(future)` to resolve a top-level future without traversing its contents (from `prefect.utilities.annotations`).
- **Task run life cycle:** Pending → Running → Completed; on error Retrying/AwaitingRetry → Running; retries exhausted → Failed; infrastructure failure → Crashed; a cached result jumps Pending → Completed. Background tasks start as `Scheduled`.

When to make something a task: it has a side effect, may fail transiently, is worth caching, or should run concurrently. **When not to:** trivial pure helpers, and anything so fine-grained that per-call overhead and UI noise exceed the value.

□□ Tasks · Retries · Run work concurrently

---

3. Task runners and concurrency control

> **Analogy: how many cooks you hire and whether they share a kitchen.** A task runner decides whether submitted tasks run in threads, processes, or on a cluster. Concurrency limits are the 'only two people at the fryer' rules.

- **Task runners** control how `.submit()`/`.map()` tasks execute: concurrent threads (the default ⚑), parallel processes, or distributed (`prefect-dask`, `prefect-ray` integrations). Set per flow with `@flow(task_runner=...)`.

- **Choose:** I/O-bound work (HTTP, DB, S3) → threads. CPU-bound Python → processes or Dask/Ray. Data too large for one machine → Dask or Ray, or push the work into the warehouse and orchestrate only.
- **Global concurrency limits** are named limits you acquire in code and share across flows and deployments, for protecting a database or an API; **rate limits** use the same mechanism with a refill rate. **Tag-based limits** cap concurrent task runs carrying a tag.

```
from prefect.concurrency.sync import concurrency, rate_limit      # ⚑ import path


@task
def write_db(rows):
    with concurrency('warehouse-writes', occupy=1):   # blocks if the limit is full
        ...
@task
def call_api(x):
    rate_limit('vendor-api')                          # waits for a slot at the configured rate
    ...
```

Traps. A task runner only affects tasks started with `.submit()` / `.map()`; direct calls are sequential. Threads do not speed up CPU-bound code. Forgetting to create the named global limit means the code has nothing to enforce. Worker-level limits (work pool and queue concurrency, §6) limit *flow runs*, not tasks.

□□ Task runners · Global concurrency limits · Tag-based limits · Per-worker task concurrency example

---

4. States, results, caching and transactions

> **Analogy: a parcel tracking page, a saved receipt and an undo button.** State says where the run is, results are the saved output, caching says 'I already did this exact job', and transactions say 'do these steps all-or-nothing'.

- **States** are rich objects (Pending, Running, Completed, Failed, Cancelled, Crashed, Paused, Scheduled, Retrying/AwaitingRetry…). They carry data and messages, drive state-change hooks and are what automations trigger on. *Crashed* means the infrastructure

failed (killed process, out of memory), not that your code raised.

- **Results** are the data a flow or task returns. Persisting them is off by default for most runs and is what enables caching and resumption. Choose `result_storage` (local path, or an S3-compatible bucket via an integration block), `persist_result=True`, and the serialiser. ⚑ Persisted results need storage reachable by every worker that may read them.
- **Caching** skips a task when a cache key matches. Policies combine with `+`: `INPUTS` (hash of arguments), `TASK_SOURCE` (the task's code), `FLOW_PARAMETERS`, `RUN_ID` and others ⚑; plus `cache_expiration`. Use a custom `key_fn` when arguments are not hashable or should be partly ignored. A cache hit moves the task straight to Completed.
- **Transactions** give tasks transactional semantics: group steps so they commit together, and register rollback hooks that run if a later step fails.

```
from prefect.transactions import transaction      # ⚑


@task
def write_staging(batch): ...
@write_staging.on_rollback
def drop_staging(txn): ...                        # runs if the transaction rolls back

@flow
def load(batch):
    with transaction():
        write_staging(batch)
        swap_into_prod()                          # if this raises, drop_staging runs
```

Use caching for expensive, deterministic steps (downloads, parsing, model scoring on fixed input) so retries and reruns are cheap. **Do not cache** steps whose result depends on time or external state unless the key includes it, or you will happily serve stale data. **Idempotency still matters:** a retry or a worker crash can run a task twice, so design writes as upserts or partition overwrites.

□□ States · Caching · Customize caching · Persist and retrieve results · Transactional workflows

---

5. Deployments: three ways to run a flow

> **Analogy: a standing order versus cooking at home.** Running the function yourself is cooking at home. A deployment is a standing order registered with the operations centre: it says what, where the recipe lives, which kitchen, and when.

Approach	How	Infrastructure	Pick it when
Run the function	`python flow.py`	Wherever you run it	Development and tests
`flow.serve()`	A long-lived Python process listens for runs and runs schedules	That process (container, VM)	Simple, single environment, no pool to manage
`flow.from_source(...).deploy()` or `prefect deploy` with `prefect.yaml`	Deployment bound to a **work pool**	Dynamically provisioned per run	Many deployments, different environments, isolation per run, platform-team guardrails

```
# 1. serve: simplest
daily_etl.serve(name='daily-etl', cron='0 6 * * *', parameters={'day': 'today'})

# 2. deploy to a work pool (code pulled from git at run time)
from prefect import flow
flow.from_source(source='https://github.com/acme/pipelines.git', entrypoint='flows/etl.py:daily_etl').deploy(
    name='daily-etl-prod', work_pool_name='k8s-pool', cron='0 6 * * *',
    parameters={'day': 'today'}, job_variables={'image': 'acme/pipelines:1.4'}, tags=['etl'])     # ⚑ argument
names

# trigger on demand
from prefect.deployments import run_deployment
run_deployment(name='daily-etl/daily-etl-prod', parameters={'day': '2026-10-22'})
```

- **Three ways to define a deployment:** Python SDK (`.deploy()`), YAML (`prefect.yaml` plus `prefect deploy`), or Terraform. All populate the same schema: name, flow, parameters, schedules, tags, work pool, job variables, version.
- **Code storage:** bake the code into the Docker image (pool types that use images), pull it from git or a cloud bucket at run time, or use a local path with Process pools. Do not rely on a path that exists only on your laptop.
- **Versioning:** deployments carry a version and can be rolled back; tag flows and tasks with `version` too. Override per-run infrastructure with job variables.
- **CI/CD:** deploy from CI on merge, one deployment per environment, with the image tag as a job variable.

Traps. A deployment does not contain code, so a worker that cannot reach the repo or image fails the run. Changing the flow signature without redeploying breaks scheduled runs that carry old

parameters. `serve` ties execution to one process that must stay up.

Deployments · Create deployments · prefect.yaml · Store flow code · Versioning · CI/CD

---

6. Work pools, workers and queues

> **Analogy: pub/sub topics.** A work pool is a topic. Deployments publish runs to it; workers subscribe and start the runs. Switch a deployment to another pool and a different worker runs it, which is how you promote between environments or debug locally.

Pool kind	Who starts runs	You run a worker?	Examples
Hybrid	A worker in your infrastructure	Yes	Process, Docker, Kubernetes, AWS ECS, Azure Container Instances, Google Cloud Run (v1, v2), Vertex AI
Push (Cloud)	Prefect submits to your serverless provider	No	ECS push, Cloud Run push, ACI push, Modal push
Managed (Cloud)	Prefect-managed infrastructure	No	Prefect Managed; Coiled is also available

- **Why pools at all:** dynamic infrastructure per run (scale to zero), default job configuration that every deployment inherits and can override, and opinionated, enforced interfaces for platform teams.
- **Workers:** `prefect worker start --pool my-pool` ⚑ polls and submits runs; a hybrid worker needs outbound access to the API and the permissions to create infrastructure. Run it as a supervised service (systemd, a Deployment) and use the worker healthcheck documented under advanced guides.
- **Work queues** sit inside a pool. Each has a **priority** (a unique positive integer; **1 is highest**) and an optional concurrency limit that cannot exceed the pool's global limit. Runs are submitted from the highest priority queue in waterfall order. A queue is `READY` if a worker polled it in the last 60 seconds, `PAUSED` when you pause it, otherwise `NOT_READY`.
- **Example policy:** low (priority 10, unlimited), high (priority 5, limit 3), critical (priority 1, limit 1) for urgent backfills.
- **Job variables and the base job template** define per-pool defaults (image, resources, env vars, namespace); override them on the deployment or even per run. Customising Kubernetes job templates has its own guide.

Pick a pool: laptop or single VM → Process. Containerised isolation on one host → Docker. Production on a cluster → Kubernetes. Serverless without operating a worker → push pool on Cloud. Avoid a Process pool for untrusted or conflicting dependencies.

Traps. The worker's own credentials decide what it can launch, so scope them. A pool with no live worker leaves runs Late. Mixing images with different Python or Prefect versions between client and worker causes confusing serialisation errors; keep versions aligned (see the versioning and compatibility page).

Work pools · Workers · Manage work pools · Kubernetes · Docker · Serverless · Job variables · Worker healthchecks

---

7. Schedules, events and automations

> **Analogy: an alarm clock versus a motion sensor.** A schedule fires at times you choose; an event trigger fires when something happens, such as a file arriving or a flow failing.

- **Schedules** tell the API to create flow runs automatically: cron, interval or RRule, attached to a deployment, with timezone support. Pause, resume or remove them per deployment.
- **Events** record activity in your stack: Prefect emits them for state changes and you can emit custom ones (`emit_event(...)` ⚑). They carry a resource, an event name and a payload.
- **Automations** pair a trigger (event, metric or compound) with actions: run a deployment, pause a schedule, send a notification, cancel a run. **Deployment triggers** make event-driven pipelines; **chaining deployments with events** gives dependencies between separate workflows without coupling the code; you can pass the event payload as flow parameters.
- **Debounce** with the `within` parameter so a burst of events does not create a run per event. **Zombie flows** (runs that stopped reporting) can be detected with an automation.

Choose a schedule for predictable batch windows; **choose an event trigger** for 'when upstream finished' or 'when data landed'. Prefer events over polling loops inside a flow.

Traps. Cron is in UTC unless a timezone is given. A schedule creates runs even if the previous one is still running unless concurrency limits stop it. Triggers that match their own outputs can loop.

Schedules · Events · Automations · Event triggers · Chaining deployments · Debouncing · Zombie flows

8. Configuration: variables, blocks, secrets, settings

Thing	Holds	Use for	Not for
Variables	Dynamically-named, mutable, non-sensitive JSON values	Shared config such as a table name or a flag	Secrets
Blocks	Typed, saved configuration with a UI (credentials, buckets, notification targets)	Reusable connections and secrets managed centrally	Per-run ephemeral data
Secret blocks	Encrypted sensitive strings	API keys, passwords	Large documents
Settings and profiles	`PREFECT_*` settings per environment	API URL, logging, timeouts, server tuning	Application config
Runtime context	Info about the current run (`prefect.runtime`)	Flow run ID, parameters, deployment name inside code	Persistent state

```
from prefect.variables import Variable
from prefect.blocks.system import Secret
from prefect.runtime import flow_run

limit = Variable.get('batch_size', default='500')
token = Secret.load('vendor-api-token').get()
print(flow_run.id, flow_run.parameters)        # ⚑ runtime attribute names
```

Switch between environments with profiles (`prefect profile use prod` ⚑) rather than editing code. Self-hosted servers have security settings (basic auth and others) documented under advanced guides; do not expose an unauthenticated API to a network.

Variables · Blocks · Settings and profiles · Runtime context · Store secrets

9. Observability, artifacts, assets and human input

- **Logging:** `log_prints=True` turns `print` into run logs; `get_run_logger()` gives a structured logger. Logging config is customisable. Cloud offers AI summaries of failed-run logs.
- **Artifacts** are persisted outputs for humans, rendered in the UI (markdown, tables, links, progress): `create_markdown_artifact(...)`, `create_table_artifact(...)` ⚑. Use them for data-quality reports and run summaries, not as a results store.
- **Assets** represent objects your workflows produce (a table, a file, a model) and are tracked as lineage across runs. Declare them on tasks with `@materialize(...)` ⚑ and add metadata and explicit dependencies where inference is not enough. Check your version supports them.
- **State change hooks** run code when a flow or task enters a state (on failure, on completion): the place for notifications and cleanup.
- **Human in the loop:** a run can *pause* and wait for input or approval, or *suspend* and free its infrastructure, then resume from the UI or API. Interactive workflows have a dedicated guide. Cancellation of running flows is a documented workflow too.

📖 Logging · Artifacts · Assets · State hooks · Interactive workflows · Cancel workflows

---

10. Background tasks

> **Analogy: a drop-box and a back-office team.** The web app drops a job in the box and returns to the customer immediately. Back-office workers pick it up whenever they are free.

`task.delay(...)` pushes a task run onto a server-side topic; a pool of **task workers** executes it. The docs position this as a replacement for Celery plus a broker for queue functionality. Use it when a web request or service must dispatch heavy work without waiting and you want to scale task execution independently. Do not use it for work that needs a flow-level schedule, deployments and per-run infrastructure; use a deployment for that. The caller does not need to resolve the returned future.

📖 Run background tasks · Web app powered by background tasks

---

11. Hosting: Cloud versus self-hosted

	Prefect Cloud	Self-hosted server
You operate	Workers and your code only	API, database, background services, workers
Extras	Push and managed pools, workspaces, SSO, service accounts, ACLs, audit logs, webhooks, SLAs, PrivateLink	Open source; fewer pool types (hybrid ones)
Limits	Documented API rate limits and data retention	Yours to size

- **Run a server:** `prefect server start` locally; Docker; Docker Compose with Postgres, Redis, server, background services and a worker; Helm chart for Kubernetes. Point clients at it with `PREFECT_API_URL`.
- **Scale:** run multiple server instances for HA (needs Postgres and Redis for messaging); keep the database maintained (monitoring, vacuuming, retention of old runs).
- **Secure:** enable authentication via the security settings, restrict network access, and see the page on endpoints and ports for restricted networks.

Server · Docker Compose · Helm · Scale self-hosted · Database maintenance · Security settings

---

12. Testing and integrations

- **Test:** a flow is a function, so unit-test the logic; the docs describe a test harness and pytest fixtures for running flows and tasks against a temporary backend. Test deployment definitions by loading them in CI.
- **Integrations** are PyPI packages: `prefect-aws` (S3, ECS worker, Secrets Manager, Lambda, Glue, Batch), `prefect-gcp`, `prefect-azure`, `prefect-kubernetes`, `prefect-docker`, `prefect-dbt` (runner with asset lineage, or per-node orchestration), `prefect-snowflake`, `prefect-sqlalchemy`, `prefect-databricks`, `prefect-dask`, `prefect-ray`, `prefect-redis`, `prefect-slack`, `prefect-email`, `prefect-github`, `prefect-gitlab`, `prefect-bitbucket`, `prefect-shell`.
- **AI assist:** a Prefect MCP server lets assistants inspect your workspace.

Test workflows · Integrations · prefect-aws · prefect-dbt · MCP server

---

13. When is Prefect the right tool?

Situation	Verdict
Python-heavy data or ML pipelines with retries, caching, scheduling, dynamic fan-out	**Good fit.** Flows are ordinary code
Event-driven pipelines (file landed, upstream finished)	**Good fit** (events and automations)
A team that wants per-run isolated infrastructure without operating a scheduler fleet	**Good fit** (work pools, push or managed)
You need a declarative, reviewable DAG with a large existing operator ecosystem and heavy SQL-first teams	Airflow may fit better; Prefect has a migration guide, so weigh the rewrite
The product is the asset graph with software-defined data assets and a data catalogue	Compare with asset-first tools; Prefect's assets are newer, so check maturity for your needs
Strict durable execution of business transactions over days with exactly-once semantics	A durable-execution engine such as Temporal is a closer match
One cron job calling one script	**Overkill.** Use cron or a platform scheduler until you need retries, visibility or dependencies
Heavy transformation inside a warehouse	Orchestrate it (dbt or SQL tasks) and let the warehouse do the work

Challenge before adopting: orchestration adds an API, a database and workers to run, so ask what failure you are actually solving (silent failures, no retries, no visibility) and whether a simpler scheduler plus alerting would do. Also confirm where results and logs may be stored if your data is regulated.

---

14. Trap checklist and exercises

1. Expecting a deployment to carry code; worker cannot fetch the repo or image.
2. Client, worker and image on different Prefect or Python versions.
3. Non-idempotent writes with retries; caching a time-dependent task with a code-only key.
4. Using threads for CPU-bound work; using `.submit()` but never resolving futures, so errors go unnoticed.
5. Dataclass fields rebuilt by parameter resolution (use `quote` or `opaque`).
6. Cron in UTC surprises; schedules overlapping long runs; trigger loops.
7. Secrets in variables instead of secret blocks; unauthenticated self-hosted API.
8. A pool with no worker, or worker credentials broader than needed.
9. Persisting results to storage that other workers cannot reach.

Exercises. (1) Write a three-task flow with `retries=3` on a flaky task and caching on an expensive one; run it twice and read the states. (2) Deploy it with `serve`, then to a Process work pool, and

compare where the code runs. (3) Add a global concurrency limit and submit 20 tasks. (4) Add a deployment trigger that runs a second flow when the first completes. (5) Wrap two writes in a `transaction()` with a rollback hook and force the second to fail.

Solution notes

1. A cached task shows Completed immediately on the second run; flow retries rerun the flow but cached tasks are skipped.
2. `serve` runs in your process; the pool path needs a worker polling the pool and a reachable code source.
3. Create the limit first (CLI or UI), then acquire it in code with `concurrency(name)`; excess tasks wait.
4. Use an automation with a flow-run-completed trigger and a run-deployment action, optionally passing the event payload.
5. The rollback hook of the first write runs, leaving no partial output.

Sources: docs.prefect.io (v3): tasks, work pools, deployments, documentation index. Other sections were written from the index summaries plus prior knowledge; items marked ⚑ should be verified.

Revision #1
Created 23 October 2026 15:06:11 by N8N
Updated 23 October 2026 15:06:11 by N8N