Prefect User Guide

Joaquín Urruti

2026-01-04

Table of contents

Prefect Workflows - User Guide

Welcome to the comprehensive Prefect usage guide in Docker. This guide will teach you everything from basic concepts to advanced configurations.

Table of Contents

8. Working with Outputs
9. Advanced Configuration
10. Practical Examples
11. Best Practices

Introduction to Prefect

Prefect is a modern workflow orchestration platform that allows you to:

- **Orchestrate** complex data pipelines
- **Schedule** automatic executions with cron
- **Monitor** the status of your workflows in real-time
- **Receive alerts** when something fails
- **Automatically retry** failed tasks
- **Register centralized logs** of all your executions

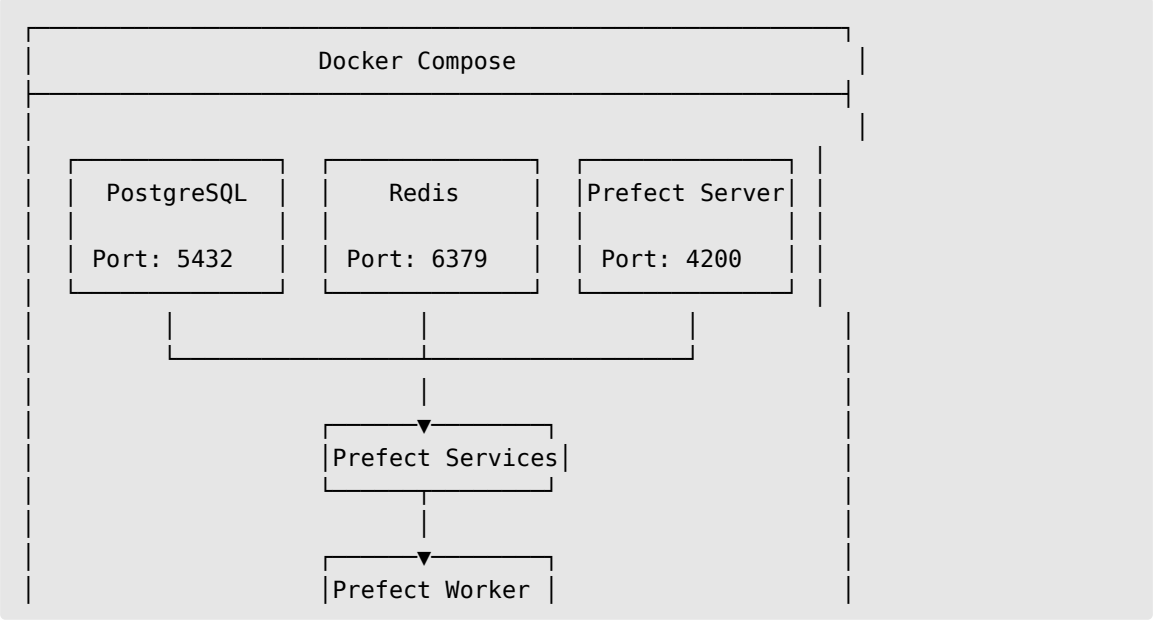
Why Prefect?

- **Simplicity:** Write normal Python code, Prefect handles the rest
- **Observability:** Modern UI for visualization and debugging
- **Reliability:** Retry system, timeouts, and error handling
- **Scalability:** From local development to distributed production

---

Project Architecture

Docker Components



```
|                    |               |                    |
|                    | Executes Flows|                    |
|                    └───────────────┘                    |
└─────────────────────────────────────────────────────────┘
```

Data Persistence

```
Host (your computer)          Docker Containers
─────────────────────         ─────────────────

./data/postgres/        ←→     PostgreSQL Data
./data/redis/           ←→     Redis Data
./logs/server/          ←→     Prefect Server Logs
./logs/services/        ←→     Prefect Services Logs
./logs/worker/          ←→     Prefect Worker Logs
./outputs/              ←→     Task Outputs
./scripts/              ←→     Python Workflows
```

Core Concepts

1. Tasks

A **task** is an individual unit of work. It's a Python function decorated with `@task`.

```python
from prefect import task

@task
def extract_data(url: str):
    """Extracts data from an API"""
    # Your code here
    return data
```

Features: - Can receive parameters - Return values - Can be automatically retried - Have integrated logging - Execute independently

2. Flows

A **flow** is a collection of orchestrated tasks. It's a Python function decorated with `@flow`.

```python
from prefect import flow, task

@task
def task_1():
    return "result 1"

@task
```

```python
def task_2(input_data):
    return f"processed: {input_data}"

@flow
def my_workflow():
    result = task_1()
    final = task_2(result)
    return final
```

Features: - Orchestrate multiple tasks - Can call other flows (subflows) - Handle state and dependencies - Can be deployed and scheduled

3. Deployments

A **deployment** is a configuration that tells Prefect how and when to execute a flow.

```python
if __name__ == "__main__":
    my_workflow.deploy(
        name="my-deployment",
        work_pool_name="local-pool",
        cron="0 9 * * *",  # Every day at 9 AM
        tags=["production", "daily"]
    )
```

4. Work Pools and Workers

- **Work Pool:** A logical group of workers
- **Worker:** The process that executes flows

In our configuration: - Work Pool: `local-pool` - Worker Type: `process` (executes in separate processes)

Creating Your First Workflow

Step 1: Create the Python File

Create a new file in `scripts/my_first_flow.py`:

```python
from prefect import flow, task
from prefect.logging import get_run_logger
import datetime

@task
def greet(name: str):
    logger = get_run_logger()
    message = f"Hello {name}!"
    logger.info(message)
    return message
```

```python
@task
def get_timestamp():
    logger = get_run_logger()
    now = datetime.datetime.now()
    logger.info(f"Timestamp: {now}")
    return now

@flow(name="My First Flow")
def my_first_flow(name: str = "World"):
    """
    A simple flow that greets and shows the time
    """
    logger = get_run_logger()
    logger.info("Starting my first flow")

    # Execute tasks
    greeting = greet(name)
    timestamp = get_timestamp()

    logger.info("Flow completed successfully")
    return {"greeting": greeting, "timestamp": timestamp}

if __name__ == "__main__":
    # Run locally for testing
    my_first_flow(name="World")
```

Step 2: Run Locally

Test your flow directly:

```
docker compose exec prefect-worker python scripts/my_first_flow.py
```

You should see the logs in the console.

Step 3: Deploy the Flow

Modify the file to add deployment:

```python
if __name__ == "__main__":
    my_first_flow.deploy(
        name="my-first-deployment",
        work_pool_name="local-pool",
        cron="*/5 * * * *",  # Every 5 minutes
        tags=["tutorial", "basic"]
    )
```

Execute the deployment:

```
docker compose exec prefect-worker python scripts/my_first_flow.py
```

Step 4: Verify in the UI

1. Open http://localhost:4200
2. Go to “Deployments”
3. You should see “my-first-deployment”
4. Go to “Flow Runs” to see executions

Working with Tasks

Task Configuration

Tasks support many configurations:

```
from prefect import task
from prefect.tasks import task_input_hash
from datetime import timedelta

@task(
    name="Process Data",
    description="Processes input data and returns results",
    tags=["processing", "data"],
    retries=3,                    # Retry 3 times if it fails
    retry_delay_seconds=60,       # Wait 60s between retries
    timeout_seconds=300,          # 5 minute timeout
    cache_key_fn=task_input_hash, # Cache results based on inputs
    cache_expiration=timedelta(hours=1),  # Cache valid for 1 hour
    log_prints=True,              # Capture prints as logs
)
def process_data(data: list):
    # Process data
    result = []
    for item in data:
        # Your logic here
        result.append(item * 2)
    return result
```

Tasks with Error Handling

```
from prefect import task, flow
from prefect.logging import get_run_logger
import requests

@task(retries=2, retry_delay_seconds=30)
def fetch_api_data(url: str):
    logger = get_run_logger()
```

```
    try:
        response = requests.get(url, timeout=10)
        response.raise_for_status()
        logger.info(f"✓ Data obtained from {url}")
        return response.json()
    except requests.RequestException as e:
        logger.error(f"✗ Error obtaining data: {e}")
        raise

@task
def process_api_data(data: dict):
    logger = get_run_logger()
    # Process the data
    result = data.get("results", [])
    logger.info(f"Processed {len(result)} items")
    return result

@flow
def api_pipeline():
    data = fetch_api_data("https://api.example.com/data")
    if data:
        return process_api_data(data)
```

Parallel Tasks

Prefect executes tasks in parallel automatically when possible:

```
from prefect import flow, task
import time

@task
def process_chunk(chunk_id: int, data: list):
    time.sleep(2)  # Simulate processing
    return f"Chunk {chunk_id}: {len(data)} items processed"

@flow
def parallel_processing():
    """
    Processes multiple chunks in parallel
    """
    chunks = [
        [1, 2, 3, 4, 5],
        [6, 7, 8, 9, 10],
        [11, 12, 13, 14, 15],
        [16, 17, 18, 19, 20]
    ]

    # These tasks run in parallel
```

```
    results = []
    for i, chunk in enumerate(chunks):
        result = process_chunk.submit(i, chunk)  # .submit() = async
        results.append(result)

    # Wait for all to complete
    return [r.result() for r in results]
```

Deployments and Scheduling

Schedule Types

1. Cron Schedule

```
@flow
def my_flow():
    pass

if __name__ == "__main__":
    my_flow.deploy(
        name="cron-example",
        work_pool_name="local-pool",
        cron="0 9 * * 1-5"  # Monday to Friday at 9 AM
    )
```

Cron Examples: - `"0 * * * *"` - Every hour - `"*/15 * * * *"` - Every 15 minutes - `"0 9,17 * * *"` - At 9 AM and 5 PM - `"0 0 * * 0"` - Sundays at midnight - `"30 2 1 * *"` - First day of the month at 2:30 AM

2. Interval Schedule

```
from prefect.client.schemas.schedules import IntervalSchedule
from datetime import timedelta

if __name__ == "__main__":
    my_flow.deploy(
        name="interval-example",
        work_pool_name="local-pool",
        interval=timedelta(hours=2)  # Every 2 hours
    )
```

3. No Schedule (Manual)

```
if __name__ == "__main__":
    my_flow.deploy(
        name="manual-example",
```

```
        work_pool_name="local-pool"
        # No cron or interval = manual execution only
    )
```

Deployment with Parameters

```
from prefect import flow

@flow
def process_file(file: str, format: str = "csv"):
    # Your code here
    pass

if __name__ == "__main__":
    process_file.deploy(
        name="process-data",
        work_pool_name="local-pool",
        parameters={
            "file": "/app/outputs/data.csv",
            "format": "csv"
        },
        cron="0 10 * * *"
    )
```

Multiple Deployments of the Same Flow

You can have different schedules for the same flow:

```
if __name__ == "__main__":
    # Deployment 1: Every hour during business hours
    my_flow.deploy(
        name="business-hours",
        work_pool_name="local-pool",
        cron="0 9-18 * * 1-5",
        parameters={"mode": "incremental"}
    )

    # Deployment 2: Daily at midnight (full refresh)
    my_flow.deploy(
        name="nightly-refresh",
        work_pool_name="local-pool",
        cron="0 0 * * *",
        parameters={"mode": "full"}
    )
```

Logging and Monitoring

Using Loggers

```python
from prefect import flow, task
from prefect.logging import get_run_logger

@task
def process_data(items: list):
    logger = get_run_logger()

    logger.debug("Starting processing")
    logger.info(f"Processing {len(items)} items")

    errors = 0
    for i, item in enumerate(items):
        try:
            # Process item
            result = item * 2
            logger.debug(f"Item {i}: {item} → {result}")
        except Exception as e:
            errors += 1
            logger.error(f"Error in item {i}: {e}")

    if errors > 0:
        logger.warning(f"Found {errors} errors")
    else:
        logger.info("✓ Processing completed without errors")

    return len(items) - errors

@flow
def my_pipeline():
    logger = get_run_logger()
    logger.info("=== Starting Pipeline ===")

    data = [1, 2, 3, 4, 5]
    processed = process_data(data)

    logger.info(f"=== Pipeline Completed: {processed} items ===")
```

Logging Levels

Logs are automatically saved in: - **Container:** `/root/.prefect/logs/` - **Host:** `./logs/worker/`

Available levels: - `DEBUG` - Detailed information for debugging - `INFO` - General flow information - `WARNING` - Warnings that don't stop execution - `ERROR` - Errors that affect the task - `CRITICAL` - Critical system errors

View Logs

From Docker

```
# Worker logs in real-time
docker compose logs -f prefect-worker

# Server logs
docker compose logs -f prefect-server

# Logs from a specific period
docker compose logs --since 1h prefect-worker
```

From the UI

1. Go to http://localhost:4200
2. Navigate to “Flow Runs”
3. Click on any run
4. Go to the “Logs” tab

From the File System

```
# View most recent logs
tail -f logs/worker/*.log

# Search for errors
grep "ERROR" logs/worker/*.log

# View logs from a specific date
ls -lh logs/worker/
```

Working with Outputs

Saving Outputs to Files

```
from prefect import flow, task
from prefect.logging import get_run_logger
import json
import csv
from pathlib import Path
from datetime import datetime

@task
def save_json(data: dict, filename: str):
    """Saves data in JSON format"""
    logger = get_run_logger()

    # Create path with timestamp
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
```

```
    filepath = Path(f"/app/outputs/{filename}_{timestamp}.json")

    # Save file
    with open(filepath, 'w', encoding='utf-8') as f:
        json.dump(data, f, indent=2, ensure_ascii=False)

    logger.info(f"✓ File saved: {filepath}")
    return str(filepath)

@task
def save_csv(data: list, filename: str):
    """Saves data in CSV format"""
    logger = get_run_logger()

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filepath = Path(f"/app/outputs/{filename}_{timestamp}.csv")

    if not data:
        logger.warning("No data to save")
        return None

    # Get headers from first row
    headers = data[0].keys()

    with open(filepath, 'w', newline='', encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=headers)
        writer.writeheader()
        writer.writerows(data)

    logger.info(f"✓ CSV saved: {filepath} ({len(data)} rows)")
    return str(filepath)

@task
def save_text_report(stats: dict, filename: str):
    """Saves a text report"""
    logger = get_run_logger()

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filepath = Path(f"/app/outputs/{filename}_{timestamp}.txt")

    with open(filepath, 'w', encoding='utf-8') as f:
        f.write("=" * 60 + "\n")
        f.write(f"REPORT GENERATED: {datetime.now()}\n")
        f.write("=" * 60 + "\n\n")

        for key, value in stats.items():
            f.write(f"{key}: {value}\n")
```

```
    logger.info(f"✓ Report saved: {filepath}")
    return str(filepath)

@flow
def pipeline_with_outputs():
    """Flow that generates multiple outputs"""
    logger = get_run_logger()

    # Generate sample data
    data = {
        "timestamp": str(datetime.now()),
        "records_processed": 1000,
        "errors": 5,
        "success_rate": 99.5
    }

    table_data = [
        {"id": 1, "name": "Item 1", "value": 100},
        {"id": 2, "name": "Item 2", "value": 200},
        {"id": 3, "name": "Item 3", "value": 300}
    ]

    stats = {
        "Total Items": len(table_data),
        "Total Sum": sum(item["value"] for item in table_data),
        "Average": sum(item["value"] for item in table_data) / len(table_data)
    }

    # Save in different formats
    json_file = save_json(data, "results")
    csv_file = save_csv(table_data, "table_results")
    txt_file = save_text_report(stats, "statistics_report")

    logger.info(f"✓ Outputs generated: JSON, CSV, TXT")

    return {
        "json": json_file,
        "csv": csv_file,
        "txt": txt_file
    }
```

Accessing the Outputs

Files are saved in `/app/outputs/` inside the container, which is mapped to `./outputs/` on your host:

```
# View generated files
ls -lh outputs/
```

```
# View JSON content
cat outputs/results_*.json | jq .

# View CSV
cat outputs/table_results_*.csv
```

Advanced Configuration

Environment Variables in Flows

```
from prefect import flow, task
import os

@task
def use_environment_variables():
    api_key = os.getenv("API_KEY", "default_key")
    env = os.getenv("ENVIRONMENT", "development")
    return f"Env: {env}, API: {api_key[:5]}..."

@flow
def flow_with_env():
    return use_environment_variables()
```

To add environment variables, edit `docker-compose.yml`:

```
prefect-worker:
  environment:
    PREFECT_API_URL: http://prefect-server:4200/api
    API_KEY: "your_api_key_here"
    ENVIRONMENT: "production"
```

Secrets and Blocks

For sensitive data, use Prefect Blocks:

```
from prefect.blocks.system import Secret

# Create secret (do once)
secret = Secret(value="my_secret_password")
secret.save("my-db-password")

# Use in a flow
@task
def connect_db():
    from prefect.blocks.system import Secret
```

```
    password = Secret.load("my-db-password").get()
    # Use password
```

Notifications

Configure notifications when a flow fails:

```
from prefect import flow
from prefect.events import emit_event

@flow
def flow_with_notifications():
    try:
        # Your code here
        result = process_data()

        # Emit success event
        emit_event(
            event="flow.completed",
            resource={"prefect.resource.id": "my_flow"}
        )

        return result
    except Exception as e:
        # Emit error event
        emit_event(
            event="flow.error",
            resource={"prefect.resource.id": "my_flow"},
            payload={"error": str(e)}
        )
        raise
```

Practical Examples

Example 1: Simple ETL

```
from prefect import flow, task
from prefect.logging import get_run_logger
import requests
import json
from datetime import datetime

@task(retries=3, retry_delay_seconds=60)
def extract_data(api_url: str):
    """Extract: Get data from an API"""
    logger = get_run_logger()
```

```
    logger.info(f"Extracting data from {api_url}")

    response = requests.get(api_url, timeout=30)
    response.raise_for_status()
    data = response.json()

    logger.info(f"✓ {len(data)} records extracted")
    return data

@task
def transform_data(data: list):
    """Transform: Clean and transform data"""
    logger = get_run_logger()
    logger.info("Transforming data")

    clean_data = []
    for item in data:
        # Transformation example
        transformed = {
            "id": item.get("id"),
            "name": item.get("name", "").upper(),
            "value": float(item.get("value", 0)),
            "processed_at": datetime.now().isoformat()
        }
        clean_data.append(transformed)

    logger.info(f"✓ {len(clean_data)} records transformed")
    return clean_data

@task
def load_data(data: list, output_file: str):
    """Load: Save processed data"""
    logger = get_run_logger()
    logger.info(f"Loading data to {output_file}")

    filepath = f"/app/outputs/{output_file}"
    with open(filepath, 'w') as f:
        json.dump(data, f, indent=2)

    logger.info(f"✓ Data loaded successfully")
    return filepath

@flow(name="ETL Pipeline")
def etl_pipeline():
    """Complete ETL pipeline"""
    logger = get_run_logger()
    logger.info("=== Starting ETL Pipeline ===")
```

```
    # Extract
    raw_data = extract_data("https://jsonplaceholder.typicode.com/users")

    # Transform
    transformed_data = transform_data(raw_data)

    # Load
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    file = load_data(transformed_data, f"etl_output_{timestamp}.json")

    logger.info("=== ETL Pipeline Completed ===")
    return file

if __name__ == "__main__":
    # Deploy with daily schedule
    etl_pipeline.deploy(
        name="daily-etl",
        work_pool_name="local-pool",
        cron="0 2 * * *",  # 2 AM every day
        tags=["etl", "production"]
    )
```

Example 2: Web Scraping

```
from prefect import flow, task
from prefect.logging import get_run_logger
import requests
from bs4 import BeautifulSoup
import csv
from datetime import datetime

@task(retries=2)
def scrape_page(url: str):
    """Scrapes a web page"""
    logger = get_run_logger()
    logger.info(f"Scraping: {url}")

    response = requests.get(url, timeout=30)
    response.raise_for_status()

    soup = BeautifulSoup(response.content, 'html.parser')

    # Example: extract titles
    titles = []
    for element in soup.find_all('h2'):
        titles.append({
            "title": element.text.strip(),
            "url": url,
```

```
            "scrape_date": datetime.now().isoformat()
        })

    logger.info(f"✓ {len(titles)} elements found")
    return titles

@task
def save_scrape_results(data: list):
    """Saves scraping results"""
    logger = get_run_logger()

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filepath = f"/app/outputs/scrape_results_{timestamp}.csv"

    if data:
        with open(filepath, 'w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=data[0].keys())
            writer.writeheader()
            writer.writerows(data)

        logger.info(f"✓ Results saved: {filepath}")

    return filepath

@flow
def web_scraping_flow(urls: list):
    """Web scraping flow"""
    logger = get_run_logger()
    logger.info(f"Starting scraping of {len(urls)} URLs")

    all_data = []
    for url in urls:
        data = scrape_page(url)
        all_data.extend(data)

    file = save_scrape_results(all_data)
    logger.info(f"✓ Scraping completed: {len(all_data)} items")

    return file
```

Example 3: File Processing

```
from prefect import flow, task
from prefect.logging import get_run_logger
import pandas as pd
from pathlib import Path

@task
```

```
def read_csv_file(filepath: str):
    """Reads a CSV file"""
    logger = get_run_logger()
    logger.info(f"Reading file: {filepath}")

    df = pd.read_csv(filepath)
    logger.info(f"✓ {len(df)} rows read")

    return df

@task
def analyze_data(df: pd.DataFrame):
    """Performs data analysis"""
    logger = get_run_logger()
    logger.info("Analyzing data")

    analysis = {
        "total_rows": len(df),
        "total_columns": len(df.columns),
        "columns": list(df.columns),
        "statistics": df.describe().to_dict()
    }

    logger.info(f"✓ Analysis completed")
    return analysis

@task
def generate_report(analysis: dict):
    """Generates analysis report"""
    logger = get_run_logger()

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filepath = f"/app/outputs/analysis_report_{timestamp}.txt"

    with open(filepath, 'w') as f:
        f.write("ANALYSIS REPORT\n")
        f.write("=" * 50 + "\n\n")
        f.write(f"Total Rows: {analysis['total_rows']}\n")
        f.write(f"Total Columns: {analysis['total_columns']}\n")
        f.write(f"\nColumns: {', '.join(analysis['columns'])}\n")

    logger.info(f"✓ Report saved: {filepath}")
    return filepath

@flow
def file_processing_flow(input_file: str):
    """Processes file and generates report"""
    logger = get_run_logger()
```

```
    logger.info("=== Starting Processing ===")

    df = read_csv_file(input_file)
    analysis = analyze_data(df)
    report = generate_report(analysis)

    logger.info("=== Processing Completed ===")
    return report
```

Best Practices

1. Code Structure

```
# ✓ GOOD: Separate concerns
@task
def extract():
    pass

@task
def transform():
    pass

@task
def load():
    pass

@flow
def etl():
    data = extract()
    clean = transform(data)
    load(clean)

# ✗ BAD: Everything in one task
@task
def do_everything():
    # Extract, transform, load all together
    pass
```

2. Appropriate Logging

```
# ✓ GOOD: Informative logging
@task
def process(items: list):
    logger = get_run_logger()
    logger.info(f"Processing {len(items)} items")
```

```
    for i, item in enumerate(items):
        logger.debug(f"Processing item {i}")
        # process

    logger.info("✓ Processing completed")

# ✗ BAD: Excessive or absent logging
@task
def process(items: list):
    for item in items:
        print(f"Item: {item}")  # Don't use print
```

3. Error Handling

```
# ✓ GOOD: Specific error handling
@task(retries=3)
def task_with_handling():
    logger = get_run_logger()
    try:
        # code
        pass
    except SpecificException as e:
        logger.error(f"Specific error: {e}")
        raise
    except Exception as e:
        logger.error(f"Unexpected error: {e}")
        raise

# ✗ BAD: Silencing errors
@task
def bad_task():
    try:
        # code
        pass
    except:
        pass  # Don't do this!
```

4. Parameterization

```
# ✓ GOOD: Use parameters
@flow
def pipeline(date: str, mode: str = "incremental"):
    # Flexible and reusable
    pass

# ✗ BAD: Hardcode values
@flow
```

```python
def pipeline():
    date = "2024-01-01"  # Hardcoded
    mode = "full"
```

5. Documentation

```python
# ✓ GOOD: Document flows and tasks
@task
def process_data(data: list) -> dict:
    """
    Processes a list of data and returns statistics.

    Args:
        data: List of dictionaries with data to process

    Returns:
        Dict with processing statistics

    Raises:
        ValueError: If data is empty
    """
    if not data:
        raise ValueError("Empty data")

    # processing
    return {"processed": len(data)}
```

Useful Reference Commands

Flow Management

```bash
# List flows
docker compose exec prefect-server prefect flow ls

# View flow details
docker compose exec prefect-server prefect flow inspect "flow-name"

# List recent runs
docker compose exec prefect-server prefect flow-run ls --limit 10
```

Deployment Management

```bash
# List deployments
docker compose exec prefect-server prefect deployment ls

# View deployment details
```

```
docker compose exec prefect-server prefect deployment inspect "flow-name/deployment-name"

# Execute a deployment manually
docker compose exec prefect-server prefect deployment run "flow-name/deployment-name"

# Pause a deployment
docker compose exec prefect-server prefect deployment pause "flow-name/deployment-name"

# Resume a deployment
docker compose exec prefect-server prefect deployment resume "flow-name/deployment-name"
```

Work Pool Management

```
# List work pools
docker compose exec prefect-server prefect work-pool ls

# View work pool details
docker compose exec prefect-server prefect work-pool inspect local-pool

# Pause a work pool
docker compose exec prefect-server prefect work-pool pause local-pool
```

Additional Resources

Official Documentation

- **Prefect Docs:** https://docs.prefect.io
- **API Reference:** https://docs.prefect.io/api-ref/
- **Concepts:** https://docs.prefect.io/concepts/

Community

- **Slack:** https://prefect.io/slack
- **GitHub:** https://github.com/PrefectHQ/prefect
- **Discourse:** https://discourse.prefect.io

Tutorials

- **Prefect Recipes:** https://docs.prefect.io/recipes/
- **Examples:** https://github.com/PrefectHQ/prefect/tree/main/examples

---

Common Troubleshooting

Flow doesn't execute on schedule

1. Verify the worker is running: `docker compose ps`
2. Verify the deployment is active in the UI
3. Check the logs: `docker compose logs -f prefect-worker`

Tasks fail with timeout

Increase the timeout in the task configuration:

```
@task(timeout_seconds=600)  # 10 minutes
def slow_task():
    pass
```

Cannot save outputs

Verify permissions of the outputs folder:

```
chmod -R 777 outputs/
```

Logs don't appear in the UI

Verify you're using `get_run_logger()`:

```
from prefect.logging import get_run_logger

@task
def my_task():
    logger = get_run_logger()  # Use this
    logger.info("Message")     # Not print()
```

You're now ready to create powerful workflows with Prefect!

For deployment, see `deploy.md`.