🚀 Prefect

Modern Workflow Orchestration

- **What is Prefect?** A modern workflow orchestration platform
- **Purpose:** Build, run, and monitor data pipelines at scale
- **Philosophy:** "Negative engineering" - eliminate workflow failures
- **Key Focus:** Developer experience and operational simplicity

"*The easiest way to coordinate your data stack*"

🛠️ Core Capabilities

Workflow Management

- Dynamic workflow generation
- Conditional branching
- Parallel execution
- Retry mechanisms

Monitoring & Observability

- Real-time dashboard
- Detailed logging
- Performance metrics
- Alert notifications

Infrastructure

- Kubernetes native
- Docker support
- Cloud integrations
- Hybrid deployments

Developer Experience

- Pure Python workflows
- Type safety
- Testing framework
- Version control integration

⚡ Quick Setup

1. Installation

```
# Install Prefect
pip install prefect

# Or with extras
pip install "prefect[dev,kubernetes]"
```

2. First Flow

```
from prefect import flow, task

@task
def say_hello(name: str) -> str:
    return f"Hello, {name}!"

@flow
def hello_world():
    greeting = say_hello("Prefect")
    print(greeting)

if __name__ == "__main__":
    hello_world()
```

3. Run It

```
python hello_flow.py
```

🔄 Dynamic Workflows

Key Features

- **Runtime Generation:** Create workflows based on data
- **Conditional Logic:** Branch based on results
- **Parameter Mapping:** Process lists dynamically
- **Subflows:** Compose complex workflows

Example: Dynamic Task Creation

```
@flow
def dynamic_workflow(items: list):
    results = []
    for item in items:
        if item > 10:
            result = process_large_item(item)
        else:
            result = process_small_item(item)
        results.append(result)
    return combine_results(results)
```

Advantage: No need to pre-define all possible workflow paths

📋 Task Management

Task Features

Execution Control

- Retries with backoff
- Timeouts
- Caching
- Concurrency limits

Resource Management

- Memory limits
- CPU requirements
- Infrastructure blocks
- Secrets management

```
@task(
    retries=3,
    retry_delay_seconds=60,
    timeout_seconds=300,
    cache_key_fn=task_input_hash,
    cache_expiration=timedelta(hours=1)
)
def robust_task(data):
    # Task implementation
    return process_data(data)
```

Task States

Pending → Running → Completed/Failed → Cached

🚀 Deployment & Scheduling

Deployment Options

Local Development

- Process-based execution
- SQLite backend
- File system storage

Production

- Kubernetes deployments
- Docker containers
- Cloud services (AWS, GCP, Azure)
- Prefect Cloud

```
# Create deployment
prefect deployment build ./flow.py:my_flow \
  --name "production-flow" \
  --schedule "0 6 * * *" \
  --work-queue "production"

# Apply deployment
prefect deployment apply my_flow-deployment.yaml

# Start agent
prefect agent start --work-queue "production"
```

Scheduling Types

- **Cron:** Traditional cron expressions
- **Interval:** Fixed time intervals
- **RRule:** Complex recurrence rules

👁️ Monitoring & Observability

Prefect UI Dashboard

Real-time Visibility

- Flow run states
- Task execution timeline
- Resource utilization
- Error tracking

Historical Analytics

- Success/failure rates
- Performance trends
- Duration metrics
- Cost analysis

Alerting & Notifications

- Slack integration
- Email notifications
- Webhook callbacks
- Custom notification blocks

```
from prefect.blocks.notifications import SlackWebhook

slack = SlackWebhook.load("my-slack-block")

@flow
def monitored_flow():
    try:
        result = risky_task()
        slack.notify("✅ Flow completed successfully")
    except Exception as e:
        slack.notify(f"❌ Flow failed: {e}")
        raise
```

🔌 Integration Ecosystem

Prefect Collections

Data Platforms

- AWS (S3, Lambda, ECS)
- GCP (BigQuery, Cloud Run)
- Azure (Blob, Functions)
- Snowflake, Databricks

Databases

- PostgreSQL, MySQL
- MongoDB, Redis
- SQLite, DuckDB

Infrastructure

- Kubernetes
- Docker
- Terraform
- GitHub Actions

ML/AI Platforms

- MLflow
- Weights & Biases
- Hugging Face
- OpenAI

```
# Example: AWS S3 integration
from prefect_aws import S3Bucket

s3_bucket = S3Bucket.load("my-bucket")

@task
def process_s3_data():
    data = s3_bucket.read_path("input/data.csv")
```

```
processed = transform_data(data)
s3_bucket.write_path("output/result.csv", processed)
```

🛡️ Error Handling & Recovery

Built-in Resilience

Automatic Recovery

- Configurable retries
- Exponential backoff
- Circuit breakers
- Graceful degradation

Error Classification

- Transient vs permanent
- Upstream dependencies
- Resource constraints
- Data quality issues

```
@task(
    retries=3,
    retry_delay_seconds=[60, 300, 900],
    retry_condition_fn=lambda task, task_run, state:
        "connection" in str(state.message).lower()
)
def resilient_api_call():
    try:
        return api_client.fetch_data()
    except ConnectionError:
        # Will be retried
        raise
    except ValidationError:
        # Won't be retried
        raise Abort("Invalid data format")
```

Recovery Strategies

- **Restart from failure:** Resume where left off
- **Skip failed tasks:** Continue with available data
- **Rollback transactions:** Maintain data consistency

- **Alternative paths:** Fallback workflows

⭐ Best Practices

Development Guidelines

Flow Design

- Keep flows focused and cohesive
- Use meaningful names
- Document parameters
- Version your flows

Task Organization

- Single responsibility principle
- Idempotent operations
- Clear input/output contracts
- Appropriate granularity

Testing Strategy

- Unit test individual tasks
- Integration test flows
- Mock external dependencies
- Validate with sample data

Production Readiness

- Comprehensive logging
- Resource monitoring
- Secrets management
- Disaster recovery plans

Golden Rule: Design for failure - assume things will go wrong and plan accordingly

Appendix

🆚 Prefect vs Apache Airflow

Aspect	Prefect	Apache Airflow
Philosophy	Negative engineering, eliminate failures	Workflow-as-code, maximum control
Dynamic Workflows	✅ Native support	⚠️ Limited, requires workarounds
Learning Curve	🟢 Gentle, Pythonic	🔴 Steep, many concepts
UI/UX	🟢 Modern, intuitive	🟡 Functional but dated
Development Speed	🟢 Fast iteration	🟡 Slower due to complexity
Ecosystem Maturity	🟡 Growing rapidly	🟢 Very mature, extensive
Community	🟡 Smaller but active	🟢 Large, established
Enterprise Features	🟢 Built-in (Prefect Cloud)	🟡 Available via Astronomer

Appendix

🚀 Advanced Features

Prefect Blocks

Configuration Management

- Reusable configuration objects
- Type-safe parameters
- Version controlled
- Environment-specific configs

```
from prefect.blocks.core import Block

class DatabaseConfig(Block):
    host: str
    port: int = 5432
    database: str

    def get_connection(self):
        return psycopg2.connect(
            host=self.host,
            port=self.port,
            database=self.database
        )
```

Work Pools & Queues

- Resource isolation
- Priority scheduling
- Automatic scaling
- Multi-cloud deployment

Artifacts & Results

- Persist task outputs
- Structured metadata
- Link to external systems

- Data lineage tracking

Subflows & Flow Composition

```
@flow
def data_ingestion_flow():
    return extract_and_validate_data()

@flow
def transformation_flow(data):
    return transform_and_enrich(data)

@flow
def main_pipeline():
    raw_data = data_ingestion_flow()
    processed_data = transformation_flow(raw_data)
    load_to_warehouse(processed_data)
```

Appendix

☁️ Prefect Cloud Features

Enterprise-Grade Platform

Collaboration

- Multi-user workspaces
- Role-based access control
- Team management
- Shared resources

Compliance & Security

- SOC 2 Type II certified
- SAML/SSO integration
- Audit logging
- Data encryption

Operational Excellence

- 99.9% uptime SLA
- Global CDN
- Automated backups
- 24/7 monitoring

Advanced Analytics

- Custom dashboards
- Cost optimization insights
- Performance analytics
- Data export APIs

Hybrid Architecture

Control Plane: Prefect Cloud manages orchestration, UI, and metadata

Data Plane: Your infrastructure executes workloads - data never leaves your environment

Pricing Tiers

- **Personal:** Free for individuals
- **Pro:** $39/month for small teams
- **Enterprise:** Custom pricing for large organizations

Appendix

🔄 Migration Strategies

From Airflow to Prefect

Assessment Phase

- Inventory existing DAGs
- Identify dependencies
- Map custom operators
- Analyze scheduling patterns

Conversion Process

- DAGs → Flows
- Tasks remain similar
- Operators → Task functions
- XComs → Return values

```
# Airflow DAG
from airflow import DAG
from airflow.operators.python import PythonOperator

dag = DAG('my_dag', schedule='@daily')

task1 = PythonOperator(
    task_id='extract',
    python_callable=extract_data,
    dag=dag
)

# Prefect Flow
from prefect import flow, task

@task
def extract_data():
    # Same function
    pass
```

```
@flow(schedule="0 0 * * *")
def my_flow():
    extract_data()
```

Migration Approaches

- **Big Bang:** Convert all workflows at once
- **Gradual:** Migrate workflows incrementally
- **Parallel:** Run both systems during transition
- **Greenfield:** Start fresh with new workflows

Pro Tip: Start with least critical workflows to gain confidence

Appendix

📈 Performance & Scalability

Scalability Patterns

Horizontal Scaling

- Multiple agents per work queue
- Distributed task execution
- Auto-scaling based on load
- Cross-region deployments

Resource Optimization

- Task concurrency limits
- Memory-efficient caching
- Lazy loading strategies
- Connection pooling

```
# Concurrency control
@task(task_run_name="process-{item}")
async def process_item(item):
    return await heavy_computation(item)

@flow(task_runner=ConcurrentTaskRunner())
async def parallel_processing():
    items = get_work_items()

    # Process up to 10 items concurrently
    results = await asyncio.gather(*[
        process_item(item)
        for item in items[:10]
    ])

    return results
```

Performance Benchmarks

- **Task Throughput:** 10,000+ tasks/minute
- **Flow Concurrency:** 1,000+ concurrent flows
- **Agent Efficiency:** <1% CPU overhead
- **Database Load:** Optimized for high-frequency updates

Real-world: Companies process 100TB+ daily with Prefect

Appendix

🧪 Testing Strategies

Testing Pyramid

Unit Tests

- Test individual tasks
- Mock external dependencies
- Validate business logic
- Fast feedback loop

```
import pytest
from unittest.mock import patch

def test_data_transformation():
    # Test task in isolation
    input_data = {"value": 100}
    result = transform_data(input_data)
    assert result["transformed_value"] == 200

@patch('external_api.fetch_data')
def test_api_task(mock_fetch):
    mock_fetch.return_value = {"status": "ok"}
    result = fetch_external_data()
    assert result["status"] == "ok"
```

Integration Tests

- Test flow execution
- Validate task interactions
- Check data flow
- Database connectivity

```
def test_data_pipeline_flow():
    # Test complete flow
    with prefect_test_harness():
        result = data_pipeline_flow()
```

```
        assert result.is_completed()

        # Check intermediate states
        task_runs = result.task_runs
        extract_run = next(
            r for r in task_runs
            if r.task_name == "extract_data"
        )
        assert extract_run.state.is_completed()
```

Test Environment Setup

```
# conftest.py
import pytest
from prefect.testing.utilities import prefect_test_harness

@pytest.fixture(autouse=True)
def prefect_test_fixture():
    with prefect_test_harness():
        yield
```

Appendix

🔒 Security Best Practices

Secrets Management

Prefect Blocks

- Encrypted storage
- Access control
- Audit logging
- Rotation support

```
from prefect.blocks.system import Secret

# Store secret securely
secret = Secret(value="my-secret-key")
secret.save("api-key")

# Use in tasks
@task
def secure_task():
    api_key = Secret.load("api-key").get()
    return call_secure_api(api_key)
```

External Secret Stores

- AWS Secrets Manager
- Azure Key Vault
- HashiCorp Vault
- Kubernetes secrets

```
from prefect_aws import AwsSecret

# Use AWS Secrets Manager
aws_secret = AwsSecret(
    aws_secret_name="prod/api-keys",
    aws_secret_key="database_password"
)
```

```
aws_secret.save("db-password")

@task
def connect_database():
    password = AwsSecret.load("db-password").get()
    return create_connection(password=password)
```

Network Security

- **VPC deployment:** Isolate Prefect infrastructure
- **TLS encryption:** All communications encrypted
- **API authentication:** Token-based access
- **Network policies:** Restrict traffic flow

Compliance Features

- **Audit trails:** Complete action logging
- **Data retention:** Configurable retention policies
- **Access reviews:** Regular permission audits
- **Incident response:** Automated alerting

Appendix

🔧 Troubleshooting Guide

Common Issues & Solutions

Flow Execution Issues

- **Tasks stuck in pending:** Check agent status
- **Import errors:** Verify dependencies
- **Memory errors:** Adjust resource limits
- **Timeout issues:** Increase task timeouts

Agent Problems

- **Agent disconnected:** Check network connectivity
- **Work queue empty:** Verify deployment status
- **Resource constraints:** Scale infrastructure

```
# Debug flow execution
@flow
def debug_flow():
    logger = get_run_logger()

    try:
        result = problematic_task()
        logger.info(f"Task succeeded: {result}")
    except Exception as e:
        logger.error(f"Task failed: {e}")
        # Create artifact for debugging
        create_flow_run_artifact(
            key="error-details",
            data={
                "error": str(e),
                "traceback": traceback.format_exc()
            }
        )
        raise
```

Debugging Tools

- **Prefect CLI:** `prefect flow-run inspect `
- **Logs:** Centralized logging in UI
- **Artifacts:** Store debug information
- **State inspection:** Examine task states

Performance Optimization

- **Task caching:** Avoid redundant computation
- **Concurrency tuning:** Balance throughput vs resources
- **Database optimization:** Index frequently queried fields
- **Memory management:** Use generators for large datasets

Appendix

📚 Resources & Community

Official Resources

Documentation

- **Docs:** docs.prefect.io
- **API Reference:** Comprehensive API docs
- **Tutorials:** Step-by-step guides
- **Examples:** Real-world use cases

Learning Paths

- **Getting Started:** Beginner tutorials
- **Advanced Patterns:** Expert techniques
- **Migration Guides:** From other tools
- **Best Practices:** Production guidelines

Community

- **Slack:** prefect.io/slack
- **GitHub:** github.com/PrefectHQ/prefect
- **Forum:** discourse.prefect.io
- **YouTube:** Video tutorials & demos

Professional Services

- **Training:** Custom workshops
- **Consulting:** Architecture guidance
- **Support:** Enterprise support plans
- **Professional Services:** Implementation help

Staying Updated

- **Blog:** prefect.io/blog - Product updates & use cases
- **Newsletter:** Monthly community newsletter
- **Webinars:** Regular technical sessions

- **Conferences:** PrefectCon annual conference

Quick Start: Try Prefect Cloud free at app.prefect.cloud

Contributing

- **Open Source:** Apache 2.0 license
- **Issues:** Bug reports & feature requests
- **PRs:** Code contributions welcome
- **Collections:** Build integrations