# Getting Started with pgwatch: Features, Installation, and Use Cases

**Pavlo Golub**
Senior Developer

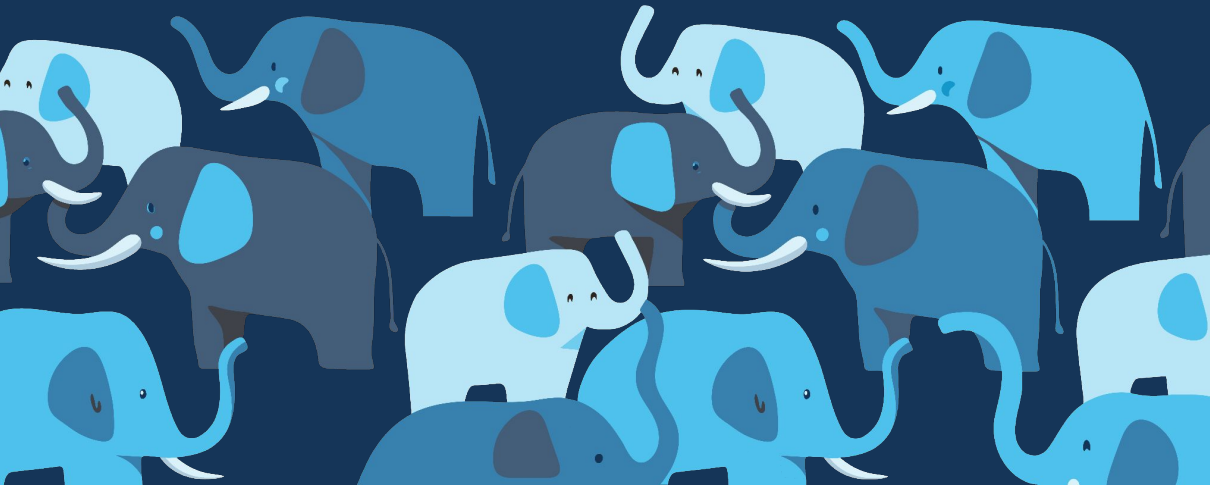
PgConf.DE 2026


CYBERTEC

# About CYBERTEC



**PostgreSQL** is at the heart of everything we do, and we are proud to actively contribute to its community and project.



## AUSTRIA (HQ)

CYBERTEC POSTGRESQL
INTERNATIONAL (HQ)

## SWITZERLAND

CYBERTEC POSTGRESQL
SWITZERLAND

## URUGUAY

CYBERTEC POSTGRESQL
SOUTH AMERICA

## ESTONIA

CYBERTEC POSTGRESQL
NORDIC

## POLAND

CYBERTEC POSTGRESQL
POLAND

## INDIA

CYBERTEC POSTGRESQL
INDIA PRIVATE LIMITED

## SOUTH AFRICA

CYBERTEC POSTGRESQL
SOUTH AFRICA

# Database Services

- 24/7 Support
- High Availability
- Consulting
- Performance Tuning
- Clustering
- Migration
- Compliance


ISO 27001 CERTIFIED

# Database Products & Tools


















# What is pgwatch?



# pgwatch at a Glance

- **Open-source** PostgreSQL monitoring solution
- Started in 2016, first release in 2017
- Mature, battle-tested in hundreds of production environments
- Written in **Go** — single binary, low resource footprint
- Collects metrics via SQL — no extensions required
- Built-in Grafana dashboards (20+ included)
- Commercially supported by CYBERTEC



# Architecture Overview

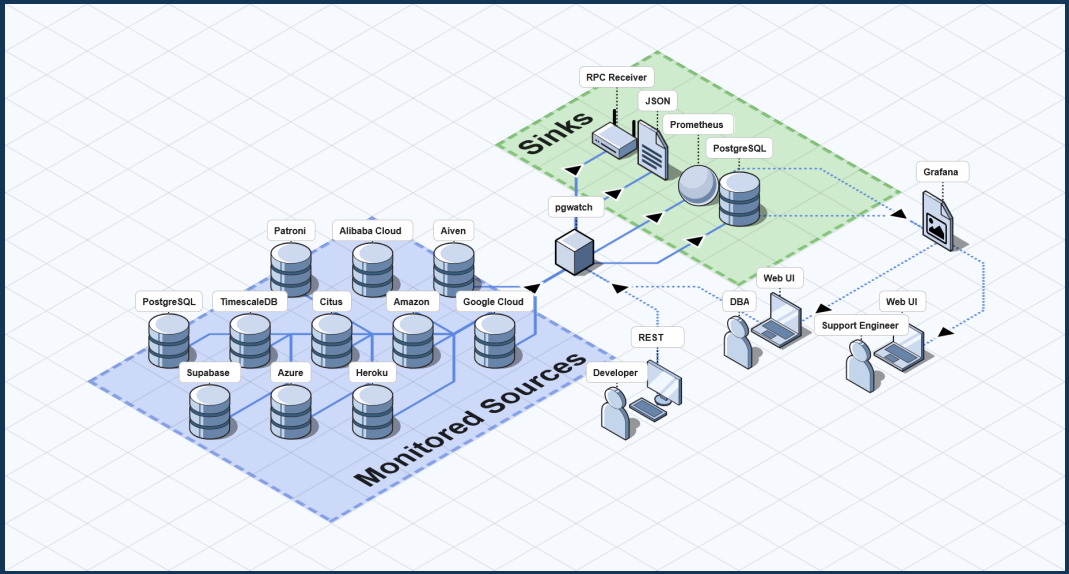
**Components:**

- **Metrics Collector** (Go binary)
- **Configuration** (YAML or PostgreSQL)
- **Metrics Storage** (sink)
- **Visualization** (Grafana)

**Sinks (storage backends):**

- PostgreSQL compatible
- Prometheus
- gRPC (custom)
- JSON files (testing)




Sinks
RPC Receiver
JSON
Prometheus
PostgreSQL
Grafana
pgwatch
Patroni
Alibaba Cloud
Aiven
PostgreSQL
TimescaleDB
Citus
Amazon
Google Cloud
Supabase
Azure
Heroku
Monitored Sources
Web UI
DBA
Web UI
Support Engineer
REST
Developer

# Key Features

- Non-invasive — no superuser or extensions needed



# Key Features

- Non-invasive — no superuser or extensions needed
- 74 built-in metrics covering all PostgreSQL subsystems



# Key Features

- Non-invasive — no superuser or extensions needed
- 74 built-in metrics covering all PostgreSQL subsystems
- 15 metric presets (minimal → exhaustive → cloud-specific)



# Key Features

- Non-invasive — no superuser or extensions needed
- 74 built-in metrics covering all PostgreSQL subsystems
- 15 metric presets (minimal → exhaustive → cloud-specific)
- Monitor PostgreSQL, PgBouncer, Pgpool, Patroni, AWS/GCP/Azure



# Key Features

- Non-invasive — no superuser or extensions needed
- 74 built-in metrics covering all PostgreSQL subsystems
- 15 metric presets (minimal → exhaustive → cloud-specific)
- Monitor PostgreSQL, PgBouncer, Pgpool, Patroni, AWS/GCP/Azure
- Continuous discovery — auto-detect new databases



# Key Features

- Non-invasive — no superuser or extensions needed
- 74 built-in metrics covering all PostgreSQL subsystems
- 15 metric presets (minimal → exhaustive → cloud-specific)
- Monitor PostgreSQL, PgBouncer, Pgpool, Patroni, AWS/GCP/Azure
- Continuous discovery — auto-detect new databases
- Custom SQL metrics — extend with your own queries



# Key Features

- Non-invasive — no superuser or extensions needed
- 74 built-in metrics covering all PostgreSQL subsystems
- 15 metric presets (minimal → exhaustive → cloud-specific)
- Monitor PostgreSQL, PgBouncer, Pgpool, Patroni, AWS/GCP/Azure
- Continuous discovery — auto-detect new databases
- Custom SQL metrics — extend with your own queries
- Web UI for configuration management + REST API



# Comparison with Other Tools

| Feature | pgwatch | pg_exporter | check_pgactivity | Datadog |
|---|---|---|---|---|
| PostgreSQL-specific | **Yes** | Yes | Yes | Partial |
| Built-in dashboards | **20+** | No | No | Yes |
| Custom SQL metrics | **Yes** | Limited | No | Limited |
| Metric presets | **15** | No | No | No |
| Cloud presets | **Yes** | No | No | Yes |
| PgBouncer/Pgpool | **Yes** | Separate | Partial | Partial |
| Web UI | **Yes** | No | No | Yes |
| Open source | **Yes** | Yes | Yes | No |
| No extensions needed | **Yes** | No | Yes | No |



# Live Demo: From Zero to Monitoring



# The Scenario

Imagine a real production environment:

- Your **DBA team** manages a PostgreSQL cluster
- Your **infra team** has Grafana running for other services
- You need to **add PostgreSQL monitoring** — fast



# The Scenario

Imagine a real production environment:

- Your **DBA team** manages a PostgreSQL cluster
- Your **infra team** has Grafana running for other services
- You need to **add PostgreSQL monitoring** — fast

Today we will:

1. Download a **single pgwatch binary**
2. Point it at our existing PostgreSQL
3. Go from zero to full monitoring in minutes
4. Detect real problems in dashboards



# The Starting Point

We have an empty working directory and **two things already running**:



# The Starting Point

We have an empty working directory and **two things already running**:

## PostgreSQL

- `localhost:5432`
- `pgwatch` role with `pg_monitor`
- `pg_stat_statements` enabled

## Grafana

- `http://localhost:3000`
- Empty — no dashboards yet
- No datasources configured

# The Starting Point

We have an empty working directory and **two things already running**:

## PostgreSQL

- `localhost:5432`
- `pgwatch` role with `pg_monitor`
- `pg_stat_statements` enabled

## Grafana

- `http://localhost:3000`
- Empty — no dashboards yet
- No datasources configured

Our job: install **one binary**, write **one YAML file**, and bring this system to life.



# Step 1 — Get the Binary



# Download pgwatch

A single Go binary. No containers, no dependencies:

```
https://github.com/cybertec-postgresql/pgwatch/releases/latest
```



# Download pgwatch

A single Go binary. No containers, no dependencies:

```
https://github.com/cybertec-postgresql/pgwatch/releases/latest
```

Or build from source:

```
git clone https://github.com/cybertec-postgresql/pgwatch
cd pgwatch/internal/webui && yarn install && yarn build && cd ../..
go build ./cmd/pgwatch/
```



# Verify the Installation

```
./pgwatch --help
```

## Expected Output

```
Version info:
  Version:       5.1.0
  Config Schema: 00824
  Sink Schema:   01180
  Git Commit:    571927698ee7b92a9f88651789c56c14cea25c21
  Built:         2026-03-17T11:31:33Z
```



# What’s Inside the Package?

Everything you need is **embedded**:

- **74 metric definitions** — SQL queries for every PostgreSQL subsystem



# What’s Inside the Package?

Everything you need is **embedded**:

- **74 metric definitions** — SQL queries for every PostgreSQL subsystem
- **15 metric presets** — from `minimal` to `exhaustive`



# What’s Inside the Package?

Everything you need is **embedded**:

- **74 metric definitions** — SQL queries for every PostgreSQL subsystem
- **15 metric presets** — from `minimal` to `exhaustive`
- **Web UI** — built-in HTTP server for configuration



# What’s Inside the Package?

Everything you need is **embedded**:

- **74 metric definitions** — SQL queries for every PostgreSQL subsystem
- **15 metric presets** — from `minimal` to `exhaustive`
- **Web UI** — built-in HTTP server for configuration
- **Schema management** — auto-creates sink tables



# What’s Inside the Package?

Everything you need is **embedded**:

- **74 metric definitions** — SQL queries for every PostgreSQL subsystem
- **15 metric presets** — from `minimal` to `exhaustive`
- **Web UI** — built-in HTTP server for configuration
- **Schema management** — auto-creates sink tables
- **Grafana dashboards** — included in the package



# What’s Inside the Package?

Everything you need is **embedded**:

- **74 metric definitions** — SQL queries for every PostgreSQL subsystem
- **15 metric presets** — from `minimal` to `exhaustive`
- **Web UI** — built-in HTTP server for configuration
- **Schema management** — auto-creates sink tables
- **Grafana dashboards** — included in the package



# What's Inside the Package?

Everything you need is **embedded**:

- **74 metric definitions** — SQL queries for every PostgreSQL subsystem
- **15 metric presets** — from `minimal` to `exhaustive`
- **Web UI** — built-in HTTP server for configuration
- **Schema management** — auto-creates sink tables
- **Grafana dashboards** — included in the package

No config files needed to start exploring:

```
./pgwatch metric list
./pgwatch metric print-sql backends --version=17
```



# Step 2 — Configure a Source



# Create `sources.yaml`

```
cat > sources.yaml << 'EOF'
- name: prod-db
  conn_str: postgresql://pgwatch:pgwatchadmin@localhost:5432/postgres
  kind: postgres
  preset_metrics: exhaustive
  is_enabled: true
EOF
```



# Create `sources.yaml`

```
cat > sources.yaml << 'EOF'
- name: prod-db
  conn_str: postgresql://pgwatch:pgwatchadmin@localhost:5432/postgres
  kind: postgres
  preset_metrics: exhaustive
  is_enabled: true
EOF
```

Key fields explained:

- **name** — unique identifier for this source
- **conn_str** — standard PostgreSQL connection string
- **kind** — `postgres`, `patroni`, `pgbouncer`, `pgpool`
- **preset_metrics** — one of 15 built-in presets
- **is_enabled** — toggle monitoring on/off



# Available Metric Presets

| Preset | Description | Metrics |
|---|---|---|
| `minimal` | Single KPI query | 1 |
| `basic` | WAL, DB size, DB stats | 5 |
| `standard` | + tables, indexes, statements | 15 |
| `exhaustive` | All important metrics | ~30 |
| `full` / `debug` | Everything (60s / 30s) | ~50–80 |
| `rds` / `aurora` / `azure` | Cloud-specific | ~25 |
| `pgbouncer` / `pgpool` | Proxy stats | ~5 |



# Test the Connection

```
./pgwatch --sources=sources.yaml source ping
```



# Test the Connection

```
./pgwatch --sources=sources.yaml source ping
```

**Expected Output**

```
OK: prod-db
```



# Step 3 — First Run: JSON Sink



# Run pgwatch with JSON Output

Start collecting metrics — write to local files first:

```
./pgwatch \
  --sources=sources.yaml \
  --sink=jsonfile://metrics-output.json
```



# Run pgwatch with JSON Output

Start collecting metrics — write to local files first:

```
./pgwatch \
  --sources=sources.yaml \
  --sink=jsonfile://metrics-output.json
```

## Expected Output

```
[INFO] [source:prod-db] [metric:wal] [interval:60] starting gatherer
[INFO] [source:prod-db] [metric:db_stats] [interval:60] starting gatherer
[INFO] [source:prod-db] [metric:backends] [interval:60] starting gatherer
[INFO] [source:prod-db] [metric:table_stats] [interval:300] starting gatherer
...
```



# Errors during the first run?

Some metrics need some preparation before they can run:

```
[ERROR] [source:prod-db] [metric:cpu_load] could not gather metric:
  ERROR: function get_load_average() does not exist
```



# Errors during the first run?

Some metrics need some preparation before they can run:

```
[ERROR] [source:prod-db] [metric:cpu_load] could not gather metric:
  ERROR: function get_load_average() does not exist
```

These metrics require extensions or stored procedures — pgwatch won't create them automatically (least-privilege principle).



# Fixing Metric Errors with `print-init`

Inspect what a metric needs, then apply it:

```
# See the init SQL for a metric
./pgwatch metric print-init cpu_load
```



# Fixing Metric Errors with `print-init`

Inspect what a metric needs, then apply it:

```
# See the init SQL for a metric
./pgwatch metric print-init cpu_load
```

## Expected Output

```
--  cpu_load
BEGIN;
CREATE EXTENSION IF NOT EXISTS plpython3u;
CREATE OR REPLACE FUNCTION get_load_average(
  OUT load_1min float, OUT load_5min float, OUT load_15min float)
AS $$ from os import getloadavg
     la = getloadavg(); return [la[0], la[1], la[2]]
$$ LANGUAGE plpython3u VOLATILE;
GRANT EXECUTE ON FUNCTION get_load_average() TO pgwatch;
COMMIT;
```



# Applying the Init SQL

Pipe the output directly to `psql` — run as a **superuser**:

```
# Apply init for one metric
export PGUSER=postgres
./pgwatch metric print-init cpu_load | psql -d mydb

# Apply init for multiple metrics (or a whole preset) at once
./pgwatch metric print-init cpu_load psutil_mem psutil_disk | psql -d mydb
./pgwatch metric print-init exhaustive | psql -d mydb
```



# Applying the Init SQL

Pipe the output directly to `psql` — run as a **superuser**:

```
# Apply init for one metric
export PGUSER=postgres
./pgwatch metric print-init cpu_load | psql -d mydb

# Apply init for multiple metrics (or a whole preset) at once
./pgwatch metric print-init cpu_load psutil_mem psutil_disk | psql -d mydb
./pgwatch metric print-init exhaustive | psql -d mydb
```

## Key Points

- `print-init` is safe to run multiple times — all statements use `IF NOT EXISTS` / `OR REPLACE`
- Run under a **superuser** account — `pgwatch` role lacks the privileges to install extensions
- Restart pgwatch (or wait for retry) after applying init SQL



# Inspect the JSON Output

```
cat metrics-output.json | jq '.' | head -25
```



# Inspect the JSON Output

```
cat metrics-output.json | jq '.' | head -25
```

## Expected Output


```
{
  "metric": "db_stats",
  "dbname": "prod-db",
  "data": [{
    "epoch_ns": 1745000000000000000,
    "numbackends": 12,
    "xact_commit": 45678,
    "blks_hit": 956789,
    "temp_bytes": 16384000,
    "deadlocks": 0
  }]
}
```




# Step 4 — Production: PostgreSQL Sink



# Bootstrap the Metrics Database

```
-- Run as superuser on your existing PostgreSQL:
CREATE ROLE pgwatch WITH
    LOGIN PASSWORD 'pgwatchadmin';

CREATE DATABASE pgwatch_metrics OWNER pgwatch;
```



# Start pgwatch with PostgreSQL Sink

```
./pgwatch \
  --sources=sources.yaml \
  --sink=postgresql://pgwatch:pgwatchadmin@localhost/pgwatch_metrics
```



# Start pgwatch with PostgreSQL Sink

```
./pgwatch \
  --sources=sources.yaml \
  --sink=postgresql://pgwatch:pgwatchadmin@localhost/pgwatch_metrics
```

## Expected Output

```
[INFO] [sink:postgres] [db:pgwatch_metrics] initialising measurements database...
[INFO] [sink:postgres] [db:pgwatch_metrics] measurements sink is activated
[INFO] [sources:1] sources refreshed
[INFO] [metrics:74] [presets:15] metrics and presets refreshed
[INFO] [source:prod-db] [recovery:false] Connect OK. Version: PostgreSQL 18.3
```



# Start pgwatch with PostgreSQL Sink

```
./pgwatch \
  --sources=sources.yaml \
  --sink=postgresql://pgwatch:pgwatchadmin@localhost/pgwatch_metrics
```

## Expected Output

```
[INFO] [sink:postgres] [db:pgwatch_metrics] initialising measurements database...
[INFO] [sink:postgres] [db:pgwatch_metrics] measurements sink is activated
[INFO] [sources:1] sources refreshed
[INFO] [metrics:74] [presets:15] metrics and presets refreshed
[INFO] [source:prod-db] [recovery:false] Connect OK. Version: PostgreSQL 18.3
```

pgwatch **auto-creates** the sink schema. No manual DDL needed.



# Verify Metrics Are Flowing

Open another terminal and check:

```
psql -U pgwatch pgwatch_metrics -c \
  "SELECT tablename FROM pg_tables
   WHERE schemaname = 'public' ORDER BY 1 LIMIT 5;"
```



# Verify Metrics Are Flowing

Open another terminal and check:

```
psql -U pgwatch pgwatch_metrics -c \
  "SELECT tablename FROM pg_tables
   WHERE schemaname = 'public' ORDER BY 1 LIMIT 5;"
```

## Expected Output

```
      tablename
----------------------
 backends
 checkpointer
 db_stats
 index_stats
 instance_up
(5 rows)
```



# Step 5 — Connect Grafana



# Add a Datasource in Grafana

Open **http://localhost:3000** → Connections → Data sources → Add



# Add a Datasource in Grafana

Open **http://localhost:3000** → Connections → Data sources → Add

| Field | Value |
|---|---|
| Type | PostgreSQL |
| Host | `localhost:5432` |
| Database | `pgwatch_metrics` |
| User / Password | `pgwatch` / `pgwatchadmin` |
| TLS/SSL Mode | disable |



# Add a Datasource in Grafana

Open **http://localhost:3000** → Connections → Data sources → Add

| Field | Value |
|---|---|
| Type | PostgreSQL |
| Host | `localhost:5432` |
| Database | `pgwatch_metrics` |
| User / Password | `pgwatch` / `pgwatchadmin` |
| TLS/SSL Mode | disable |

Click **Save & Test** — should show “Database Connection OK”.



# Import Dashboards

Dashboards are included in the release package (`zip`, `deb`, `rpm`).



# Import Dashboards

Dashboards are included in the release package (`zip`, `deb`, `rpm`).

In Grafana: **Dashboards** → **New** → **Import**

1. Click **Upload dashboard JSON file**
2. Pick files from `grafana/postgres/v12/` in the package
3. Select the datasource you just created
4. Click **Import**



# Import Dashboards

Dashboards are included in the release package (`zip`, `deb`, `rpm`).

In Grafana: **Dashboards** → **New** → **Import**

1. Click **Upload dashboard JSON file**
2. Pick files from `grafana/postgres/v12/` in the package
3. Select the datasource you just created
4. Click **Import**

Repeat for each dashboard — or use provisioning in production:

```
cp grafana/dashboards.yml /etc/grafana/provisioning/dashboards/
cp -r grafana/postgres/v12 /var/lib/grafana/dashboards/postgres
systemctl restart grafana-server
```



# Built-in Dashboards

pgwatch ships with **20+ Grafana dashboards**:

- Global DB Overview
- DB Overview
- Query Performance Analysis
- Tables Overview
- Table Details
- Index Overview
- Sessions Overview
- Replication
- System Stats
- Checkpointer/BGWriter
- Sproc Details
- PgBouncer Stats
- Recommendations
- Server Log Events



# Built-in Dashboards

pgwatch ships with **20+ Grafana dashboards**:

- Global DB Overview
- DB Overview
- Query Performance Analysis
- Tables Overview
- Table Details
- Index Overview
- Sessions Overview
- Replication
- System Stats
- Checkpointer/BGWriter
- Sproc Details
- PgBouncer Stats
- Recommendations
- Server Log Events

Default home dashboard: **Global DB Overview**



# Dashboard: Global DB Overview

This dashboard shows across **all monitored databases**:

- Connection count and utilization
- Transaction rates (commits vs rollbacks)
- Tuple operations (inserts, updates, deletes)
- Temporary files and bytes
- Database sizes over time
- WAL generation rate
- Deadlock count
- Cache hit ratio



# Dashboard: Query Performance Analysis

- Top queries by total time
- Top queries by calls
- Top queries by mean time
- Slowest individual queries
- Query plan changes over time
- I/O timing breakdown



# Dashboard: Query Performance Analysis

- Top queries by total time
- Top queries by calls
- Top queries by mean time
- Slowest individual queries
- Query plan changes over time
- I/O timing breakdown

**Requires**

```
shared_preload_libraries = 'pg_stat_statements'
pg_stat_statements.track = all
```



# Step 6 — Explore the Web UI



# Web UI Overview

pgwatch includes a built-in Web UI — no extra install:

Open: **http://localhost:8080**



# Web UI Overview

pgwatch includes a built-in Web UI — no extra install:

Open: **http://localhost:8080**

The Web UI provides:

- Source management (add/edit/remove monitored databases)
- Metric configuration and preset selection
- Status overview of all monitored sources
- Log viewer



# Web UI: Sources Management

From the Web UI you can:

- Add new sources with a form
- Test connections before saving
- Toggle monitoring on/off
- Switch presets per source
- Set custom metric intervals



# Adding a Source via Web UI

**Required fields:**

- Unique name
- Connection string
- Kind (postgres, patroni, etc.)
- Metric preset or custom config

**Optional fields:**

- Include/exclude DB patterns
- Custom tags (e.g., `env: prod`)
- Statement timeout
- Standby-specific metrics



# Step 7 — Custom Metrics



# Creating a Custom Metric

**Start from the built-in metrics** — then append your own:

```
# Export all 74 built-in metrics as your base
./pgwatch metric list > metrics.yaml

# Append your custom metric definition
cat >> metrics.yaml << 'EOF'
  slow_queries:
    sqls:
      11: |
        SELECT /* pgwatch_generated */
          (extract(epoch from now()) * 1e9)::int8 as epoch_ns,
          count(*)::int as total_slow,
          max(extract(epoch from now()-query_start))::float as max_s
        FROM pg_stat_activity
        WHERE state = 'active'
          AND now()-query_start > interval '5 seconds'
          AND pid != pg_backend_pid();
    gauges: ['*']
    is_instance_level: true
    node_status: primary
EOF
```



# Important: `--metrics` Replaces Everything

**Warning**

`--metrics` **replaces** built-in metrics entirely.
A file with only `slow_queries` means pgwatch knows **nothing else**.



# Important: `--metrics` Replaces Everything

**Warning**

`--metrics` **replaces** built-in metrics entirely.
A file with only `slow_queries` means pgwatch knows **nothing else**.
Always start from `./pgwatch metric list > metrics.yaml` and **append** your custom definitions.



# Using Custom Metrics in Sources

Drop `preset_metrics` and list metrics explicitly:

```yaml
- name: prod-db
  conn_str: postgresql://pgwatch:pgwatchadmin@localhost/postgres
  kind: postgres
  custom_metrics:
    backends: 60
    db_stats: 60
    wal: 60
    table_stats: 300
    slow_queries: 15   # our custom metric
  is_enabled: true
```



# Using Custom Metrics in Sources

Drop `preset_metrics` and list metrics explicitly:

```
- name: prod-db
  conn_str: postgresql://pgwatch:pgwatchadmin@localhost/postgres
  kind: postgres
  custom_metrics:
    backends: 60
    db_stats: 60
    wal: 60
    table_stats: 300
    slow_queries: 15   # our custom metric
  is_enabled: true
```

## Key Points

- `custom_metrics` maps metric name → interval in seconds
- `preset_metrics` **overrides** `custom_metrics` — use one or the other
- All metric names must exist in the loaded `--metrics` file

# Start with Custom Metrics

```
./pgwatch \
  --sources=sources.yaml \
  --metrics=metrics.yaml \
  --sink=postgresql://pgwatch:pgwatchadmin@localhost/pgwatch_metrics
```



# Start with Custom Metrics

```
./pgwatch \
  --sources=sources.yaml \
  --metrics=metrics.yaml \
  --sink=postgresql://pgwatch:pgwatchadmin@localhost/pgwatch_metrics
```

Your `metrics.yaml` contains **all 74 built-in + your custom metric**. pgwatch loads the full set — nothing is lost.



# What a Metric Looks Like Inside

```
./pgwatch metric print-sql backends --version=17
```

## Expected Output

```
-- backends
with sa_snapshot as (
  select * from pg_stat_activity
  where pid != pg_backend_pid()
  and datname = current_database()
)
select /* pgwatch_generated */
  (extract(epoch from now()) * 1e9)::int8 as epoch_ns,
  (select count(*) from sa_snapshot where backend_type = 'client backend') as total,
  ...
```



# Problem Solving: Live Dashboards



# The Chaos We Created

Our `demo-chaos-load.sh` script has been running for 30 minutes.

It created **8 problem streams** — let's find them all in the dashboards:

- Slow sequential scans on 2M-row unindexed table



# The Chaos We Created

Our `demo-chaos-load.sh` script has been running for 30 minutes.

It created **8 problem streams** — let's find them all in the dashboards:

- Slow sequential scans on 2M-row unindexed table
- Table bloat on `demo_hot` (autovacuum disabled!)



# The Chaos We Created

Our `demo-chaos-load.sh` script has been running for 30 minutes.

It created **8 problem streams** — let's find them all in the dashboards:

- Slow sequential scans on 2M-row unindexed table
- Table bloat on `demo_hot` (autovacuum disabled!)
- Temp file spills from large sorts with `work_mem = 1MB`



# The Chaos We Created

Our `demo-chaos-load.sh` script has been running for 30 minutes.

It created **8 problem streams** — let's find them all in the dashboards:

- Slow sequential scans on 2M-row unindexed table
- Table bloat on `demo_hot` (autovacuum disabled!)
- Temp file spills from large sorts with `work_mem = 1MB`
- Long-running idle-in-transaction sessions (2 min holds)



# The Chaos We Created

Our `demo-chaos-load.sh` script has been running for 30 minutes.

It created **8 problem streams** — let's find them all in the dashboards:

- Slow sequential scans on 2M-row unindexed table
- Table bloat on `demo_hot` (autovacuum disabled!)
- Temp file spills from large sorts with `work_mem = 1MB`
- Long-running idle-in-transaction sessions (2 min holds)
- Lock contention with 5 queued waiters



# The Chaos We Created

Our `demo-chaos-load.sh` script has been running for 30 minutes.

It created **8 problem streams** — let's find them all in the dashboards:

- Slow sequential scans on 2M-row unindexed table
- Table bloat on `demo_hot` (autovacuum disabled!)
- Temp file spills from large sorts with `work_mem = 1MB`
- Long-running idle-in-transaction sessions (2 min holds)
- Lock contention with 5 queued waiters
- Deadlocks — two sessions updating rows in opposite order



# The Chaos We Created

Our `demo-chaos-load.sh` script has been running for 30 minutes.

It created **8 problem streams** — let's find them all in the dashboards:

- Slow sequential scans on 2M-row unindexed table
- Table bloat on `demo_hot` (autovacuum disabled!)
- Temp file spills from large sorts with `work_mem = 1MB`
- Long-running idle-in-transaction sessions (2 min holds)
- Lock contention with 5 queued waiters
- Deadlocks — two sessions updating rows in opposite order
- Connection spikes — 30 connections every 4 minutes



# The Chaos We Created

Our `demo-chaos-load.sh` script has been running for 30 minutes.

It created **8 problem streams** — let's find them all in the dashboards:

- Slow sequential scans on 2M-row unindexed table
- Table bloat on `demo_hot` (autovacuum disabled!)
- Temp file spills from large sorts with `work_mem = 1MB`
- Long-running idle-in-transaction sessions (2 min holds)
- Lock contention with 5 queued waiters
- Deadlocks — two sessions updating rows in opposite order
- Connection spikes — 30 connections every 4 minutes
- Unused and duplicate indexes



# Dashboard: Global DB Overview

Open Grafana: **http://localhost:3000**



# Dashboard: Global DB Overview

Open Grafana: **http://localhost:3000**

You should immediately see:

- **Deadlock count** — incrementing every 5 minutes
- **Connection spikes** — periodic jumps of +30 backends
- **Temp files/bytes** — spikes from large sorts
- **Transaction rate** — steady baseline from pgbench
- **Rollback rate** — deadlock-related rollbacks



# Detecting Slow Queries

Switch to the **Query Performance Analysis** dashboard.

The chaos script runs a **full seq scan** and a **disk-spilling join** every 30s:

```
SELECT count(*), avg(amount) FROM demo_slow
  WHERE bloat_text LIKE '%deadbeef%';

SELECT count(*) FROM demo_slow a
  JOIN demo_slow b ON a.hash_data = b.hash_data
  WHERE a.id < 50000;
```



# Detecting Slow Queries

Switch to the **Query Performance Analysis** dashboard.

The chaos script runs a **full seq scan** and a **disk-spilling join** every 30s:

```
SELECT count(*), avg(amount) FROM demo_slow
  WHERE bloat_text LIKE '%deadbeef%';

SELECT count(*) FROM demo_slow a
  JOIN demo_slow b ON a.hash_data = b.hash_data
  WHERE a.id < 50000;
```

## What to look for

- `mean_exec_time` > 200ms for both queries
- `shared_blks_read` >> `shared_blks_hit` — cold cache reads
- `temp_blks_written` > `0` — the self-join spills to disk
- These queries dominate the **top by total time** list



# Dashboard: Recommendations

The **recommendations** preset checks for:

- **Unused indexes** — wasting disk and slowing writes



# Dashboard: Recommendations

The **recommendations** preset checks for:

- **Unused indexes** — wasting disk and slowing writes
- **Duplicate indexes** — redundant definitions



# Dashboard: Recommendations

The **recommendations** preset checks for:

- **Unused indexes** — wasting disk and slowing writes
- **Duplicate indexes** — redundant definitions
- **Tables needing VACUUM/ANALYZE**



# Dashboard: Recommendations

The **recommendations** preset checks for:

- **Unused indexes** — wasting disk and slowing writes
- **Duplicate indexes** — redundant definitions
- **Tables needing VACUUM/ANALYZE**
- **Default settings** that should be tuned



# Dashboard: Recommendations

The **recommendations** preset checks for:

- **Unused indexes** — wasting disk and slowing writes
- **Duplicate indexes** — redundant definitions
- **Tables needing VACUUM/ANALYZE**
- **Default settings** that should be tuned
- **Security** — unencrypted connections, weak auth



# Dashboard: Recommendations

The **recommendations** preset checks for:

- **Unused indexes** — wasting disk and slowing writes
- **Duplicate indexes** — redundant definitions
- **Tables needing VACUUM/ANALYZE**
- **Default settings** that should be tuned
- **Security** — unencrypted connections, weak auth
- **Bloated tables** — high dead tuple ratio



# Dashboard: Recommendations

The **recommendations** preset checks for:

- **Unused indexes** — wasting disk and slowing writes
- **Duplicate indexes** — redundant definitions
- **Tables needing VACUUM/ANALYZE**
- **Default settings** that should be tuned
- **Security** — unencrypted connections, weak auth
- **Bloated tables** — high dead tuple ratio
- **Sprocs with high total time**



# Dashboard: Recommendations

The **recommendations** preset checks for:

- **Unused indexes** — wasting disk and slowing writes
- **Duplicate indexes** — redundant definitions
- **Tables needing VACUUM/ANALYZE**
- **Default settings** that should be tuned
- **Security** — unencrypted connections, weak auth
- **Bloated tables** — high dead tuple ratio
- **Sprocs with high total time**
- **Sequences approaching overflow**



# Advanced: Scaling Up



# Multiple Sinks Simultaneously

pgwatch supports **multiple sinks** at once:

```
./pgwatch \
  --sources=sources.yaml \
  --sink=postgresql://pgwatch:pgwatchadmin@localhost/pgwatch_metrics \
  --sink=prometheus://0.0.0.0:9187/pgwatch
```



# Multiple Sinks Simultaneously

pgwatch supports **multiple sinks** at once:

```
./pgwatch \
  --sources=sources.yaml \
  --sink=postgresql://pgwatch:pgwatchadmin@localhost/pgwatch_metrics \
  --sink=prometheus://0.0.0.0:9187/pgwatch
```

```
curl -s http://localhost:9187/metrics | head -10
```

```
# HELP pgwatch_db_stats_numbackends Number of backends
# TYPE pgwatch_db_stats_numbackends gauge
pgwatch_db_stats_numbackends{dbname="prod-db"} 12
pgwatch_db_stats_xact_commit{dbname="prod-db"} 45678
```



# Environment Variable Substitution

Sources support **environment variable** expansion:

```
- name: $MY_SOURCE_NAME
  conn_str: $DATABASE_URL
  kind: $SOURCE_KIND
  preset_metrics: $PRESET
  is_enabled: true
  group: $GROUP
  include_pattern: $INCLUDE_DBS
  exclude_pattern: $EXCLUDE_DBS
```



# Environment Variable Substitution

Sources support **environment variable** expansion:

```yaml
- name: $MY_SOURCE_NAME
  conn_str: $DATABASE_URL
  kind: $SOURCE_KIND
  preset_metrics: $PRESET
  is_enabled: true
  group: $GROUP
  include_pattern: $INCLUDE_DBS
  exclude_pattern: $EXCLUDE_DBS
```

## Use Case

- **CI/CD pipelines** — inject connection strings at deploy time
- **Kubernetes** — use secrets and config maps
- **Multi-environment** — same YAML, different env vars per stage



# Continuous Discovery

For dynamic environments, use `postgres-continuous-discovery`:

```
- name: prod-cluster
  conn_str: postgresql://pgwatch@prod-primary:5432/postgres
  kind: postgres-continuous-discovery
  preset_metrics: standard
  is_enabled: true
  include_pattern: "^(app_|analytics_)"
  exclude_pattern: "^template"
```



# Continuous Discovery

For dynamic environments, use `postgres-continuous-discovery`:

```
- name: prod-cluster
  conn_str: postgresql://pgwatch@prod-primary:5432/postgres
  kind: postgres-continuous-discovery
  preset_metrics: standard
  is_enabled: true
  include_pattern: "^(app_|analytics_)"
  exclude_pattern: "^template"
```

pgwatch will **automatically discover** and monitor:

- All databases matching the include pattern
- New databases added after pgwatch starts
- Databases are removed from monitoring when dropped



# Patroni Cluster Monitoring

```
- name: patroni-cluster
  kind: patroni
  conn_str: etcd://etcd1:2379,etcd2:2379/service/myapp
  preset_metrics: exhaustive
  preset_metrics_standby: standard
  is_enabled: true
  only_if_master: false
```



# Patroni Cluster Monitoring

```
- name: patroni-cluster
  kind: patroni
  conn_str: etcd://etcd1:2379,etcd2:2379/service/myapp
  preset_metrics: exhaustive
  preset_metrics_standby: standard
  is_enabled: true
  only_if_master: false
```

- Discovers cluster members via **etcd**, **Consul**, or **ZooKeeper**
- Automatically follows **primary/standby** role changes
- Different metric presets for primary vs standby
- Works with `only_if_master: true` to monitor only the primary



# Running as a systemd Service

For production, run pgwatch as a system service:

```
[Unit]
Description=pgwatch PostgreSQL monitoring
After=network.target

[Service]
Type=simple
User=pgwatch
ExecStart=/usr/local/bin/pgwatch \
  --sources=/etc/pgwatch/sources.yaml \
  --sink=postgresql://pgwatch@localhost/pgwatch_metrics
Restart=always

[Install]
WantedBy=multi-user.target
```

```
systemctl enable --now pgwatch
journalctl -u pgwatch -f
```



# Summary



# What We Built Today

**From zero:**

- Downloaded one binary
- Wrote one YAML file
- Tested with JSON sink
- Connected to PostgreSQL sink
- Imported Grafana dashboards

**To full monitoring:**

- 15+ metrics every 10–60s
- Slow query detection
- Lock contention analysis
- Table bloat alerts
- Automated recommendations



# What We Built Today

**From zero:**

- Downloaded one binary
- Wrote one YAML file
- Tested with JSON sink
- Connected to PostgreSQL sink
- Imported Grafana dashboards

**Total files created:** 1 (`sources.yaml`)

**Total infrastructure changes:** 0

**To full monitoring:**

- 15+ metrics every 10–60s
- Slow query detection
- Lock contention analysis
- Table bloat alerts
- Automated recommendations



# Getting Started Checklist

1. Download the pgwatch binary
2. Create `sources.yaml` with your databases
3. `./pgwatch source ping --sources=sources.yaml`
4. Start with `--sink=jsonfile://test.json` to verify
5. Switch to `--sink=postgresql://...` for production
6. Import Grafana dashboards from the package
7. Enable `pg_stat_statements` for query analysis



# Getting Started Checklist

1. Download the pgwatch binary
2. Create `sources.yaml` with your databases
3. `./pgwatch source ping --sources=sources.yaml`
4. Start with `--sink=jsonfile://test.json` to verify
5. Switch to `--sink=postgresql://...` for production
6. Import Grafana dashboards from the package
7. Enable `pg_stat_statements` for query analysis

## Resources

- GitHub: `github.com/cybertec-postgresql/pgwatch`
- Docs: `pgwat.ch`
- Docker Hub: `cybertecpostgresql/pgwatch`



# Thank You

**Questions?**

github.com/cybertec-postgresql/pgwatch