

Extending RTLola with External Data Queries

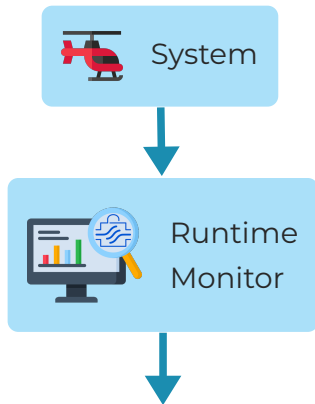
*Bernd Finkbeiner¹, Jakob Hirschler²,
Frederik Scheerer¹, Sebastian Schirmer²*

¹CISPA, ²German Aerospace Center (DLR)

RV 2026 | October 7, 2026

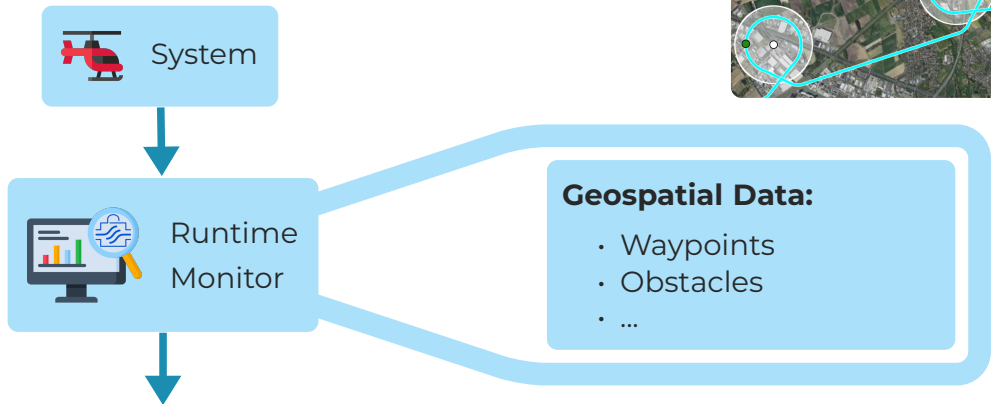


Motivation



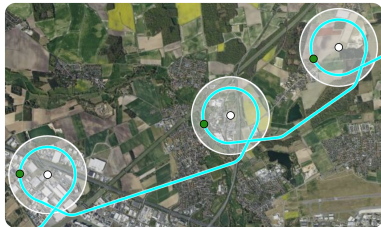


Motivation





Motivation



System



Runtime
Monitor

Stream-Based Monitors:

Efficient Evaluation of
Temporal Operators

Geospatial Data:

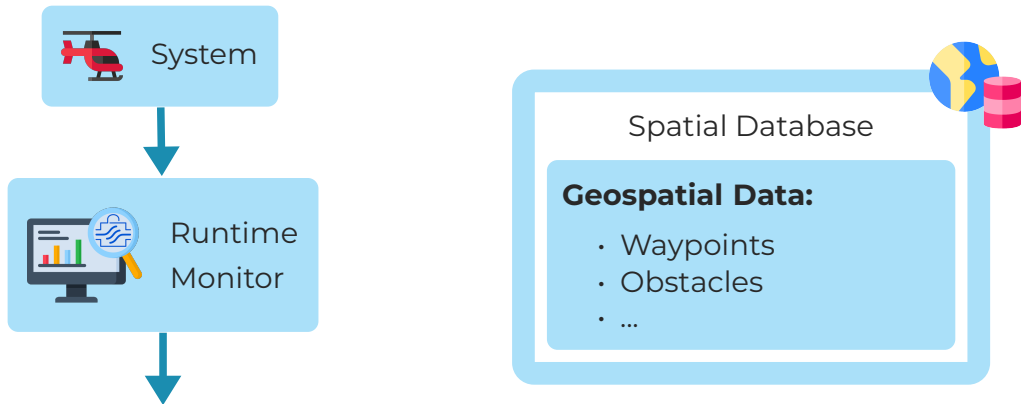
- Waypoints
- Obstacles
- ...

Spatial Databases:

Highly Optimized for
Spatial Queries

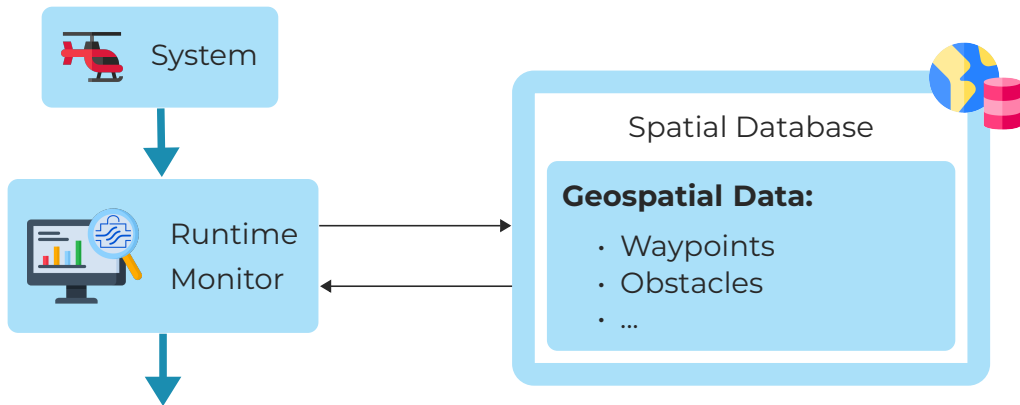


Motivation





Motivation

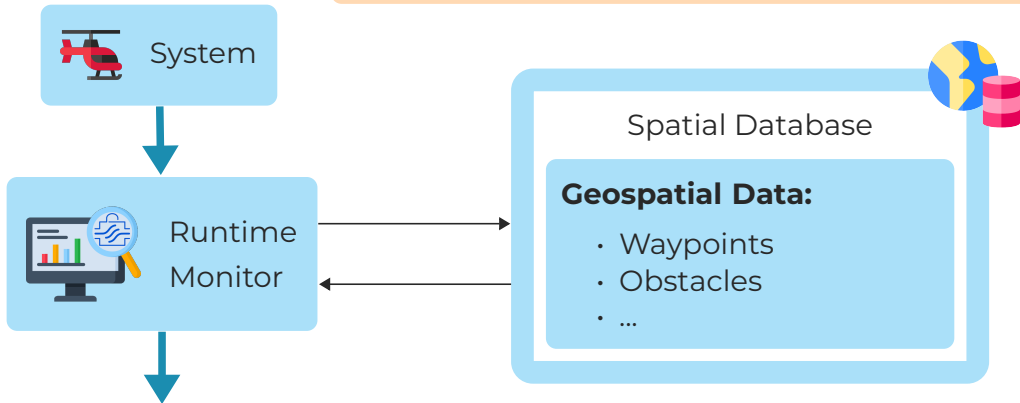


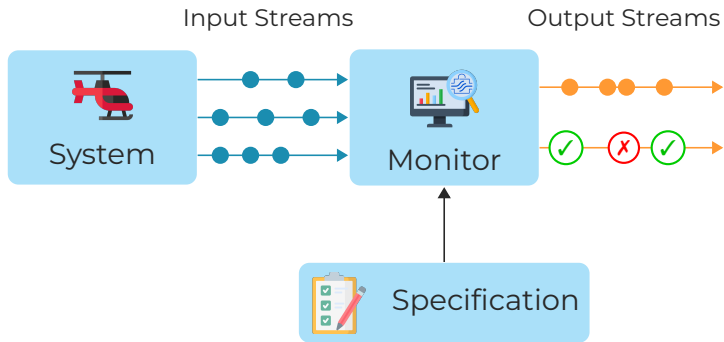


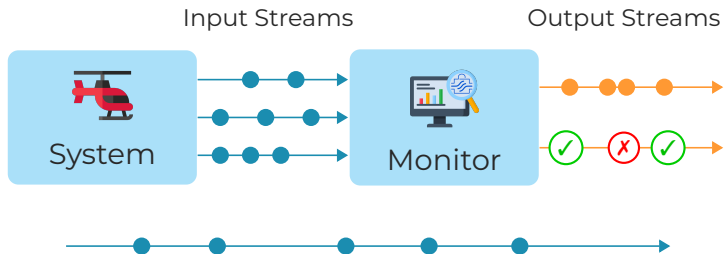
Motivation

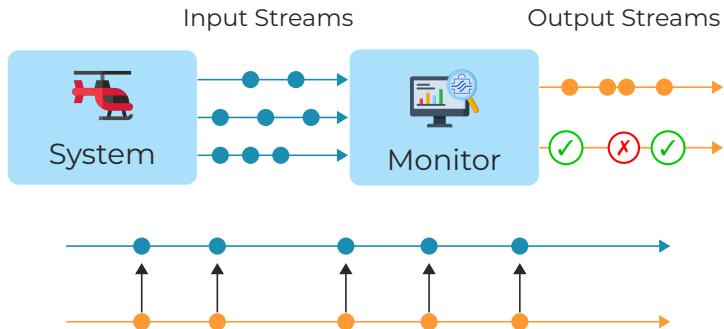
Our contribution: We extend RTLola so specifications can actively query external data sources at runtime.

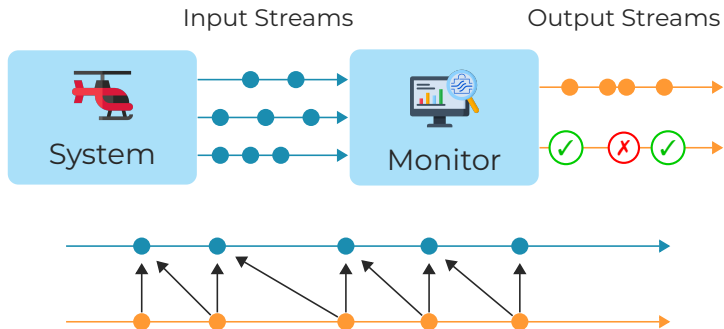
✓ Non-blocking ✓ Type-safe ✓ Fault-tolerant ✓ Backend-agnostic

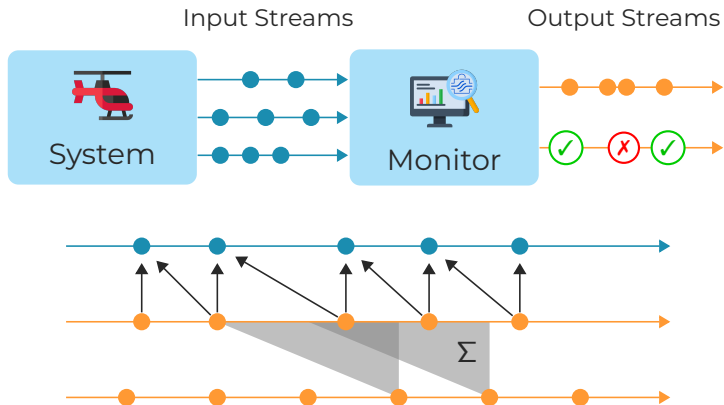














Challenges

Multiple Backends



- Relational databases
- Geospatial databases
- External APIs
- Different protocols and formats



Challenges

Multiple Backends



- Relational databases
- Geospatial databases
- External APIs
- Different protocols and formats

Asynchronous Queries



- Queries may have high latency
- Monitor must not block
- Continue unrelated evaluations



Challenges

Multiple Backends



- Relational databases
- Geospatial databases
- External APIs
- Different protocols and formats

Asynchronous Queries



- Queries may have high latency
- Monitor must not block
- Continue unrelated evaluations

Type Safety



- Validate returned data
- Ensure expected types
- Prevent runtime errors



Challenges

Multiple Backends



- Relational databases
- Geospatial databases
- External APIs
- Different protocols and formats

Asynchronous Queries



- Queries may have high latency
- Monitor must not block
- Continue unrelated evaluations

Type Safety



- Validate returned data
- Ensure expected types
- Prevent runtime errors

Error Handling



- Queries may fail
- Monitor must not crash
- Detect failures
- Define fallback behavior



Challenges

Multiple Backends



- Relational databases
- Geospatial databases
- External APIs
- Different protocols and formats

Asynchronous Queries



- Queries may have high latency
- Monitor must not block
- Continue unrelated evaluations

Type Safety



- Validate returned data
- Ensure expected types
- Prevent runtime errors

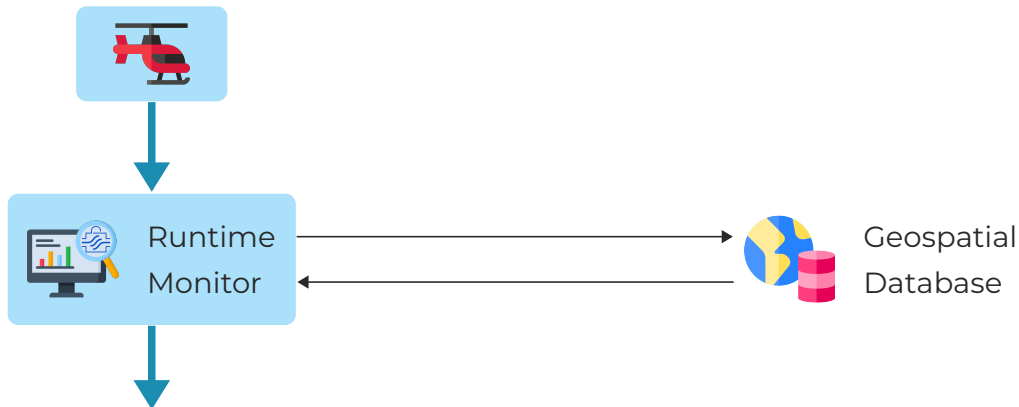
Error Handling



- Queries may fail
- Monitor must not crash
- Detect failures
- Define fallback behavior

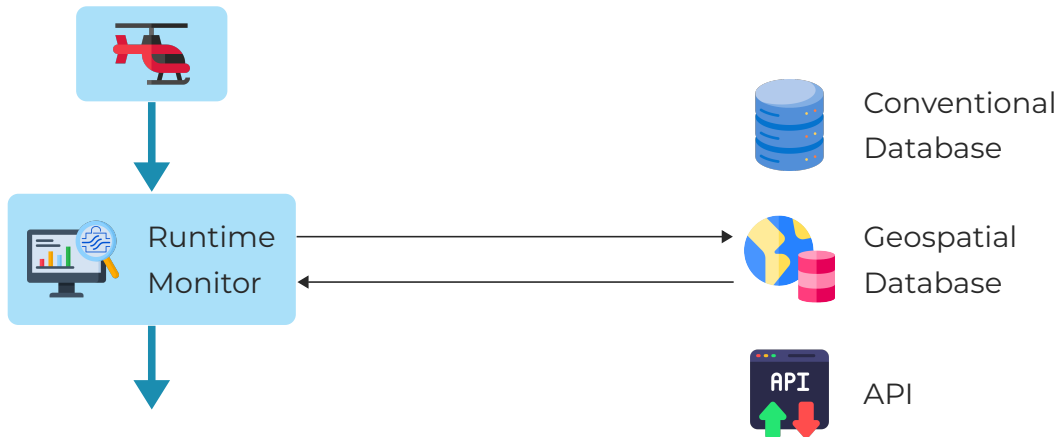


Multiple Backends



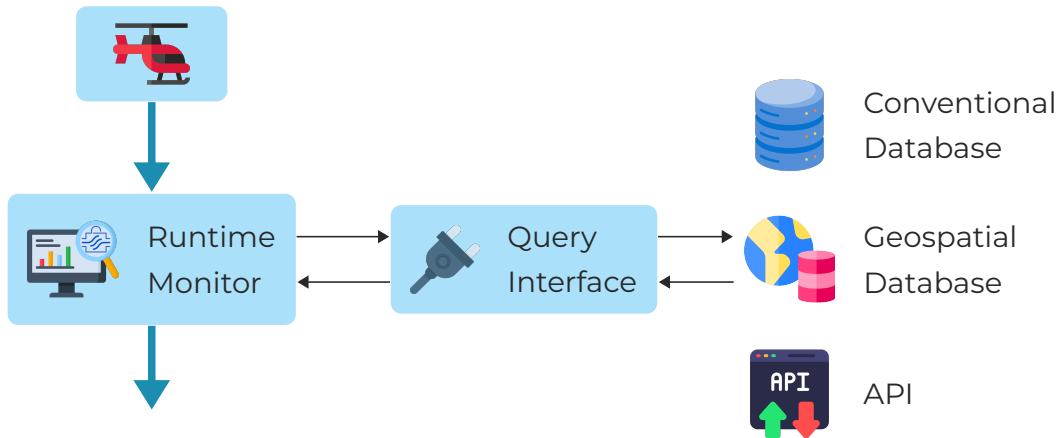


Multiple Backends



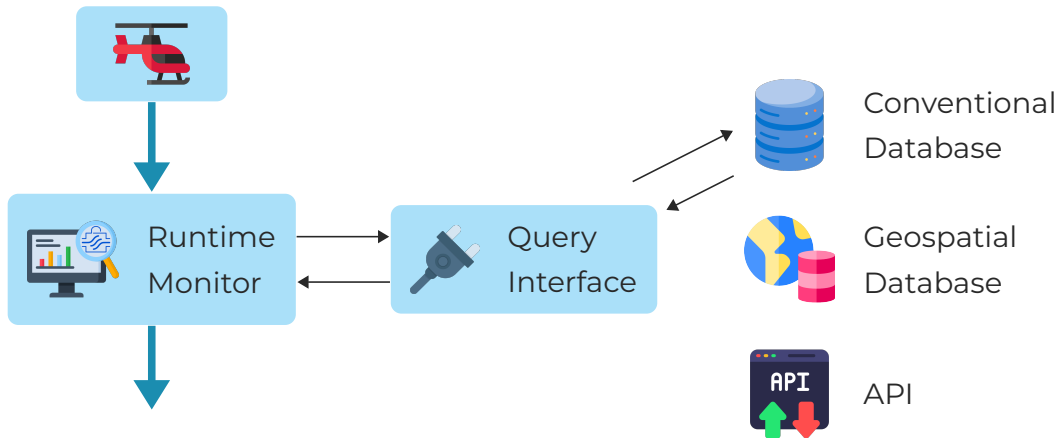


Multiple Backends





Multiple Backends





Challenges

Multiple Backends



- Relational databases
- Geospatial databases
- External APIs
- Different protocols and formats

Asynchronous Queries



- Queries may have high latency
- Monitor must not block
- Continue unrelated evaluations

Type Safety



- Validate returned data
- Ensure expected types
- Prevent runtime errors

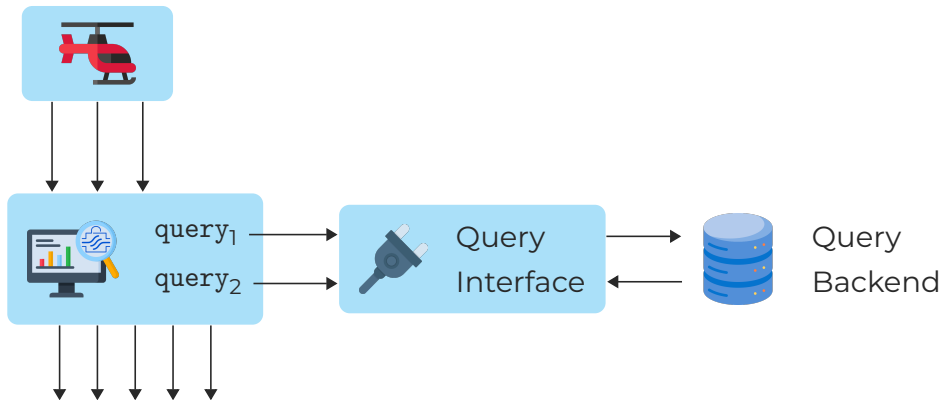
Error Handling



- Queries may fail
- Monitor must not crash
- Detect failures
- Define fallback behavior

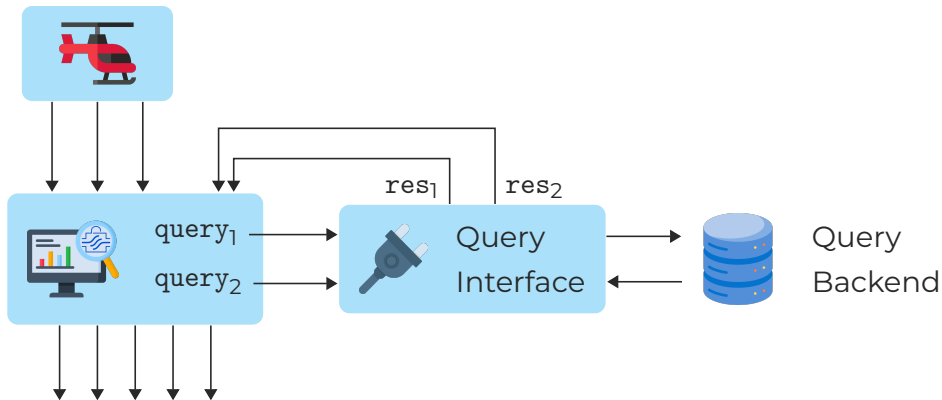


Overview





Overview





Example

```
input pos : (Float64, Float64)
```

For data transmission a celltower must be in reach **continuously for 30 seconds**.

Such a transmission period must happen **at least every 5 minutes**.



Example

```
input pos : (Float64, Float64)
```

For data transmission a celltower must be in reach **continuously for 30 seconds**.
Such a transmission period must happen **at least every 5 minutes**.

```
output queries @pos := query("NEARBY celltowers COUNT POINT ?1 ?2 700",  
                             with: pos, into: ct_res)
```



Example

For data transmission a celltower must be in reach **continuously for 30 seconds**.
Such a transmission period must happen **at least every 5 minutes**.

```
input pos : (Float64, Float64)
```

Count number of celltowers
closer than 700m from (?1, ?2)

```
output queries @pos := query("NEARBY celltowers COUNT POINT ?1 ?2 700",  
                               with: pos, into: ct_res)
```



Example

For data transmission a celltower must be in reach **continuously for 30 seconds**.

Such a transmission period must happen **at least every 5 minutes**.

```
input pos : (Float64, Float64)
```

Count number of celltowers
closer than 700m from (?1, ?2)

```
output queries @pos := query("NEARBY celltowers COUNT POINT ?1 ?2 700",  
                               with: pos, into: ct_res)
```

Instantiate (?1, ?2)
with current location



Example

For data transmission a celltower must be in reach **continuously for 30 seconds**.
Such a transmission period must happen **at least every 5 minutes**.

```
input pos : (Float64, Float64)
```

```
input ct_res : (QueryId, UInt64)
```

Count number of celltowers
closer than 700m from (?1, ?2)

```
output queries @pos := query("NEARBY celltowers COUNT POINT ?1 ?2 700",  
    with: pos, into: ct_res)
```

Instantiate (?1, ?2)
with current location

Stream results
into ct_res



Example

For data transmission a celltower must be in reach **continuously for 30 seconds**.
Such a transmission period must happen **at least every 5 minutes**.

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
```

Count number of celltowers
closer than 700m from (?1, ?2)

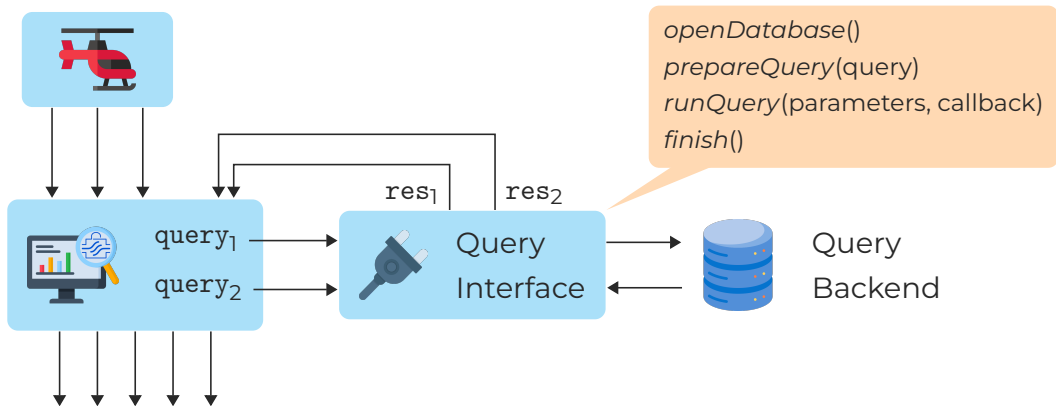
```
output queries @pos := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                             with: pos, into: ct_res)
```

Instantiate (?1, ?2)
with current location

Stream results
into ct_res

```
output in_range := ct_res.1 > 0
output in_range_s @1s := in_range.aggregate(over: 30s, using: forall)
trigger @1s ¬in_range_s.aggregate(over: 5min, using: exists)
```

Overview





Challenges

Multiple Backends



- Relational databases
- Geospatial databases
- External APIs
- Different protocols and formats

Asynchronous Queries



- Queries may have high latency
- Monitor must not block
- Continue unrelated evaluations

Type Safety



- Validate returned data
- Ensure expected types
- Prevent runtime errors

Error Handling



- Queries may fail
- Monitor must not crash
- Detect failures
- Define fallback behavior



Type Checking

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
```

```
output queries @pos := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                             with: pos, into: ct_res)
```

```
output in_range := ct_res.1 > 0
output in_range_s @1s := in_range.aggregate(over: 30s, using: forall)
trigger @1s ¬in_range_s.aggregate(over: 5min, using: exists)
```



Type Checking

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
```



Query supported by backend

```
output queries @pos := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
    with: pos, into: ct_res)
```

```
output in_range := ct_res.1 > 0
output in_range_s @1s := in_range.aggregate(over: 30s, using: forall)
trigger @1s ¬in_range_s.aggregate(over: 5min, using: exists)
```



Type Checking

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
```



Query results match input type

```
output queries @pos := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
    with: pos, into: ct_res)
```

```
output in_range := ct_res.1 > 0
output in_range_s @1s := in_range.aggregate(over: 30s, using: forall)
trigger @1s ¬in_range_s.aggregate(over: 5min, using: exists)
```

Type Checking

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
```

```
output queries @pos := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                             with: pos, into: ct_res)
```



Placeholder matches with-expression

```
output in_range := ct_res.1 > 0
output in_range_s @1s := in_range.aggregate(over: 30s, using: forall)
trigger @1s ¬in_range_s.aggregate(over: 5min, using: exists)
```



Challenges

Multiple Backends



- Relational databases
- Geospatial databases
- External APIs
- Different protocols and formats

Asynchronous Queries



- Queries may have high latency
- Monitor must not block
- Continue unrelated evaluations

Type Safety



- Validate returned data
- Ensure expected types
- Prevent runtime errors

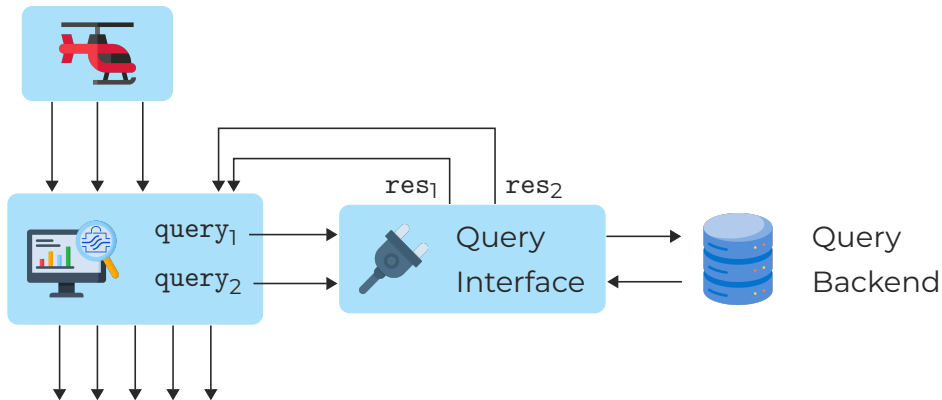
Error Handling



- Queries may fail
- Monitor must not crash
- Detect failures
- Define fallback behavior

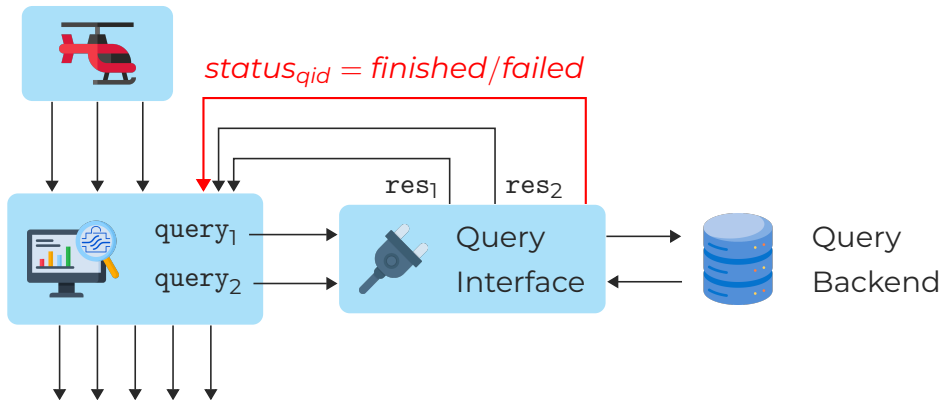


Overview





Overview





Error Handling Example

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
output queries @pos : QueryId := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                                     with: pos, into: ct_res)
```



Error Handling Example

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
output queries @pos : QueryId := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                                     with: pos, into: ct_res)
```



Error Handling Example

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
output queries @pos : QueryId := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                                       with: pos, into: ct_res)

#[query_status]
input query_status : (QueryId, QueryStatus) as (status_id, status_msg)
```



Error Handling Example

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
output queries @pos : QueryId := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                                       with: pos, into: ct_res)

#[query_status]
input query_status : (QueryId, QueryStatus) as (status_id, status_msg)
trigger @query_status status_msg == QueryStatus::Error "query error"
```



Error Handling Example

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
output queries @pos : QueryId := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                                       with: pos, into: ct_res)

#[query_status]
input query_status : (QueryId, QueryStatus) as (status_id, status_msg)
trigger @query_status status_msg == QueryStatus::Error "query error"

input time : Float64
output query_time(qid)
  spawn with queries
  eval when queries == qid with time
  close @query_status when status_id == qid
```



Error Handling Example

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
output queries @pos : QueryId := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                                       with: pos, into: ct_res)

#[query_status]
input query_status : (QueryId, QueryStatus) as (status_id, status_msg)
trigger @query_status status_msg == QueryStatus::Error "query error"

input time : Float64
output query_time(qid)
  spawn with queries
    eval when queries == qid with time
    close @query_status when status_id == qid
output query_delay @query_status := time - query_time(status_id).hold(or: 0.0)
```



Error Handling Example

```
input pos : (Float64, Float64)
input ct_res : (QueryId, UInt64)
output queries @pos : QueryId := query("NEARBY celltowers COUNT POINT ?1 ?2 700",
                                       with: pos, into: ct_res)

#[query_status]
input query_status : (QueryId, QueryStatus) as (status_id, status_msg)
trigger @query_status status_msg == QueryStatus::Error "query error"

input time : Float64
output query_time(qid)
  spawn with queries
    eval when queries == qid with time
    close @query_status when status_id == qid
output query_delay @query_status := time - query_time(status_id).hold(or: 0.0)
trigger query_delay > 1.0 "long query delay"
```

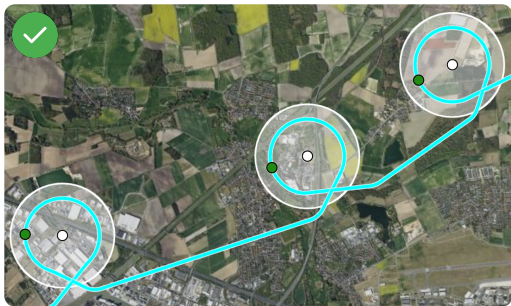


Celltower Monitoring



For data transmission a celltower must be in reach **continuously for 30 seconds**.

Such a transmission period must happen **at least every 5 minutes**.



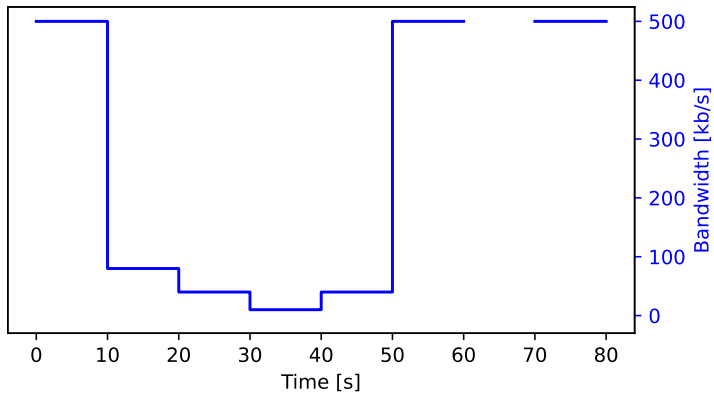


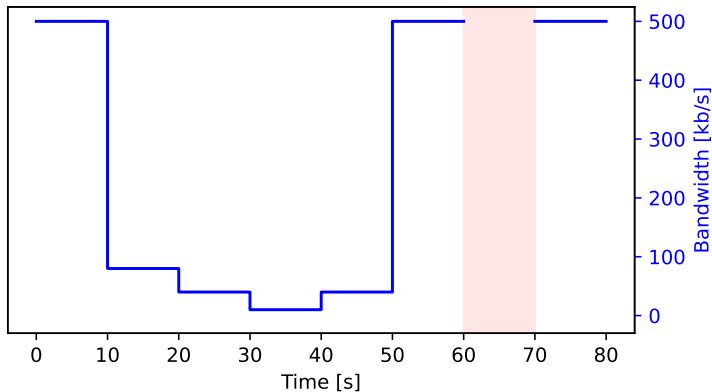
Benchmark

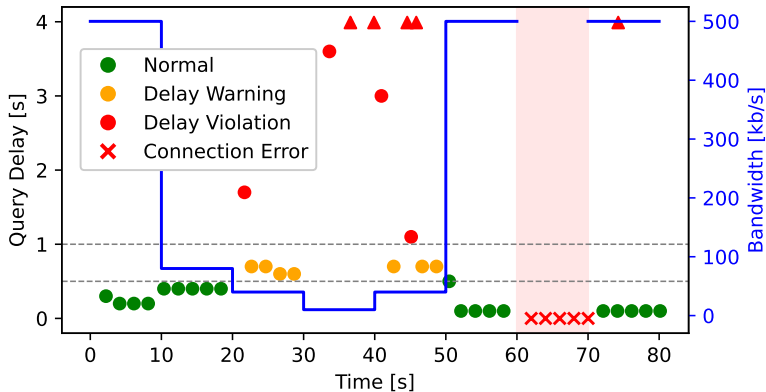


Backend	Purpose	Runtime (# Powerlines)		
		10k	100k	1.7m
RTLola	Baseline	876ms	24.4s	-
RTLola + SQLite3	General-purpose	1.04s	11.6s	250s
RTLola + Tile38	Geospatial	4.0ms	3.9ms	4.3ms
RTLola + Our kd-tree	Geospatial	2.8ms	2.7ms	3.0ms

Each trace with 190 queries.









Conclusion

Our contribution: We extend RTLola so specifications can actively query external data sources at runtime.

✓ Non-blocking ✓ Type-safe ✓ Fault-tolerant ✓ Backend-agnostic



Our contribution: We extend RTLola so specifications can actively query external data sources at runtime.

✓ Non-blocking ✓ Type-safe ✓ Fault-tolerant ✓ Backend-agnostic

Classical Monitoring

One verdict
at end of trace

Stream-Based Monitoring

Multiple verdicts
Rich, (asynchronous)
data streams

Stream-Based + Data Queries

Actively queries
its environment



Our contribution: We extend RTLola so specifications can actively query external data sources at runtime.

✓ Non-blocking ✓ Type-safe ✓ Fault-tolerant ✓ Backend-agnostic

Classical Monitoring

One verdict
at end of trace

Stream-Based Monitoring

Multiple verdicts
Rich, (asynchronous)
data streams

Stream-Based + Data Queries

Actively queries
its environment

Future Work: Large language models as a query backend:
monitor can reason over unstructured data.

The icons used in this presentation are designed by Magnific:
www.magnific.com.