Kasadara

Observability in LLM Applications

LangSmith & OpenTelemetry

Chinnasamy

Kasadara

Blind Spots in LLM Observability

OPAQUE DECISION PATHS

LLMs don't explain why they chose a specific output.

UNTRACKED PROMPT AND RESPONSE FLOW

Inputs, intermediate steps, and outputs often go unlogged.

TOOL INVOCATION IS HIDDEN

External tools used in chains or agents lack visibility.

NO INSIGHTS INTO ERROR

Failures and incorrect responses are difficult to trace or evaluate.

OpenTelemetry

Kasadara

WHY OpenTelemetry and LangSmith

OpenTelemetry – Unified Observability

- **Open standard** for collecting **traces, metrics, and logs**
- Vendor-neutral → no vendor lock-in
- Enables **end-to-end visibility** across distributed systems
- Supported by major cloud platforms (AWS, Azure, GCP)

LangSmith – LLM Debugging & Evaluation

- Purpose-built for **LLM application tracing and evaluation**
- **Integrates with OpenTelemetry** to track LangChain workflows
- Offers **fine-grained insight** into prompts, model responses, and tool usage
- Facilitates **prompt engineering**, **performance tuning**, and **error analysis**

OpenTelemetry

Kasadara

OpenTelemetry

PLANNING

Modern applications are becoming increasingly distributed — spanning across microservices, APIs, and third-party tools.

METHOD

Traditional observability relies on fragmented logs and isolated tools.

CRITERIA

A modern observability solution must:

- Capture logs, metrics, and traces
- Be language-agnostic and vendor-neutral
- Work seamlessly in distributed & GenAI architectures

An open-source observability framework providing a unified approach to collect, process, and export telemetry data — enabling full visibility across your application stack.

OpenTelemetry

Kasadara

OpenTelemetry Setup

```
C:\Users\Asus\Downloads\opentelemetry-movies-microservices-main>docker run -d -p 9411:9411 openzipkin/zipkin
609745f29fa1e5511dc6a23046ae0c0339378901cdf17acee3e5aa619c867738
```

Creation of docker container

```
PS C:\Users\Asus\development\opentelemetry> npm install @opentelemetry/api@^1.8.0 `
>> @opentelemetry/auto-instrumentations-node@^0.45.0 `
>> @opentelemetry/exporter-metrics-otlp-proto@^0.51.0 `
>> @opentelemetry/exporter-trace-otlp-proto@^0.51.0 `
>> @opentelemetry/sdk-metrics@^1.24.0 `
>> @opentelemetry/sdk-node@^0.51.0 `
>> @opentelemetry/sdk-trace-node@^1.24.0 `
>> express@^4.19.2
```

Install of all necessary packages:

- Opentelemetry api
- exporter - otlp
- Sdk - metrics
- Trace - node

```
npm warn ERESOLVE overriding peer dependency
npm warn deprecated acorn-import-assertions@1.9.0: package has been renamed to acorn-import-attributes
npm warn deprecated @opentelemetry/plugin-https@0.16.0: Deprecated in favor of @opentelemetry/instrumentation-http
npm warn deprecated @opentelemetry/plugin-http@0.16.0: Deprecated in favor of @opentelemetry/instrumentation-http
npm warn deprecated @opentelemetry/node@0.16.0: Package renamed to @opentelemetry/sdk-trace-node
npm warn deprecated @opentelemetry/tracing@0.16.0: Package renamed to @opentelemetry/sdk-trace-base
npm warn deprecated @opentelemetry/plugin-express@0.13.1: Deprecated in favor of @opentelemetry/instrumentation-express
npm warn deprecated @opentelemetry/api-metrics@0.24.0: Please use @opentelemetry/api >= 1.3.0
npm warn deprecated @opentelemetry/metrics@0.24.0: Package renamed to @opentelemetry/sdk-metrics-base

added 374 packages, and audited 375 packages in 30s

20 packages are looking for funding
  run `npm fund` for details

6 vulnerabilities (3 low, 3 high)
```

OpenTelemetry

Kasadara

OpenTelemetry Workflow

Instrumentation

Code is wrapped to emit telemetry data

Data Capture

Spans, logs, metrics collected at runtime

Processing

Collector enriches, batches, and routes data

Exporting

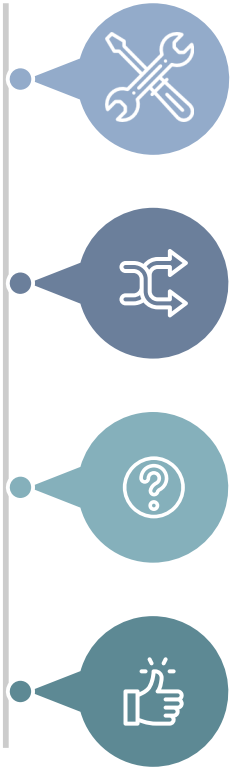
Data sent to backend tools

Monitoring

You observe flows, errors, and latencies

OpenTelemetry

How OpenTelemetry Captures System context



Instrumentation Hooks

OpenTelemetry SDKs hook into frameworks (e.g., HTTP, gRPC, DB) and automatically attach system metadata to each span.

DARING

Each request/operation carries context (like trace ID, user/session data) via HTTP headers (W3C Trace Context). This ensures continuity across microservices.

ANSWER

The OpenTelemetry Collector can be configured to inject additional metadata, such as host info, region, or deployment environment.

CONCLUSION

Developers can manually add custom key-value pairs to spans/logs (e.g., prompt_id, user_type, agent_name) for LLM-specific context.

OpenTelemetry

Kasadara

Kasadara

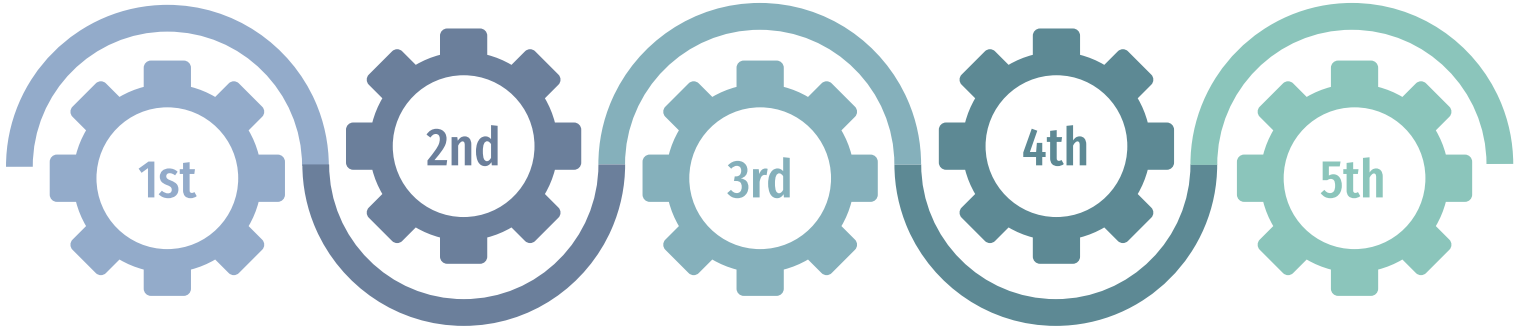
Implementation of OTel on projects

COMBINING MODEL WITH THE OTEL

Enable context tracking between services and model calls

METRICS

Monitor resource usage and model performance



SETTING UP

Prepare your project and environment

TRACING

Track model inference and system behavior

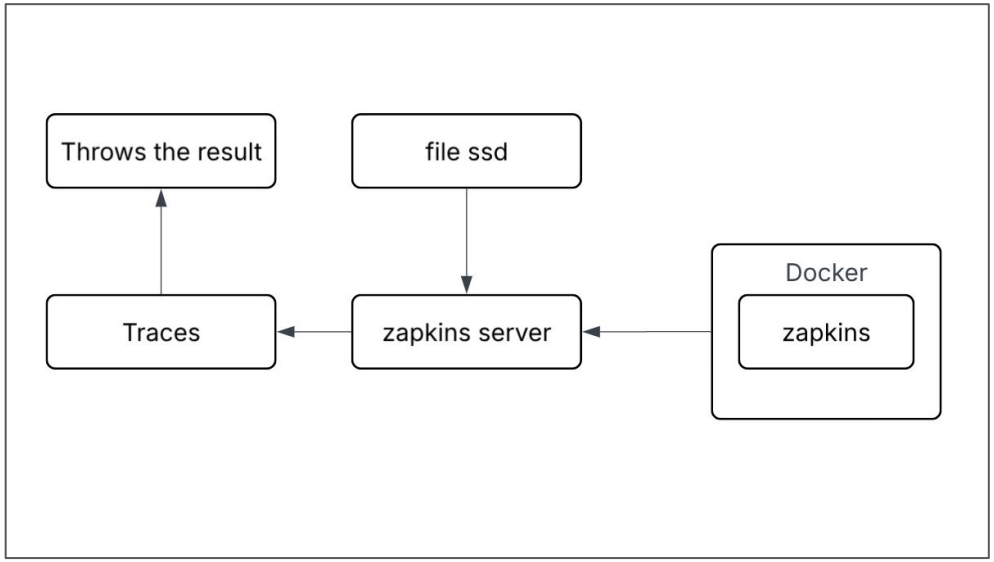
ANALYSIS TOOLS

Visualize and analyze with observability platforms

OpenTelemetry

Kasadara

Work Flow of Tracing using OTel

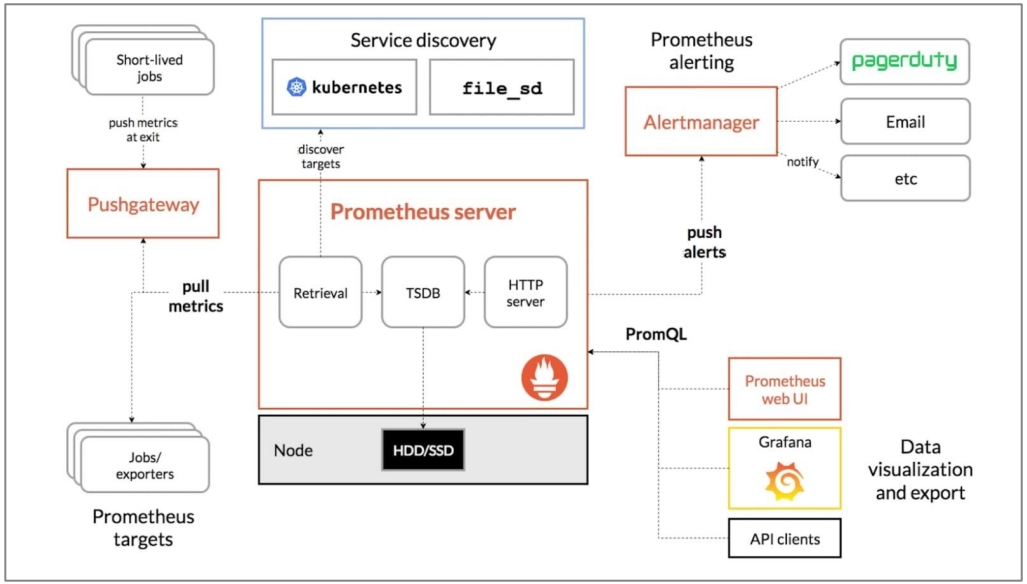


- **Docker runs Zapkins** to collect trace data.
- **Zapkins server** receives and manages trace data.
- Trace data is stored on **file SSD** (local storage).
- **Traces** module gets data from Zapkins server.
- Finally, it **throws the result** (outputs the processed trace).

OpenTelemetry

Kasadara

Work Flow of Metric using OTel



Prometheus Server: Pulls metrics from targets, stores them in TSDB, and exposes them via HTTP API.

Targets (Jobs/Exporters): Provide metrics over HTTP; Prometheus scrapes them at intervals.

Pushgateway: Allows short-lived jobs to push metrics; Prometheus pulls from Pushgateway.

Service Discovery: Auto-discovers targets using integrations like **Kubernetes** and **file_sd**.

Alertmanager: Receives alerts from Prometheus, manages routing, deduplication, and sends notifications (e.g., PagerDuty, Email).

PromQL: Query language for selecting and aggregating time series data.

OpenTelemetry

Kasadara

Tracing the Model

tracing.js > ...

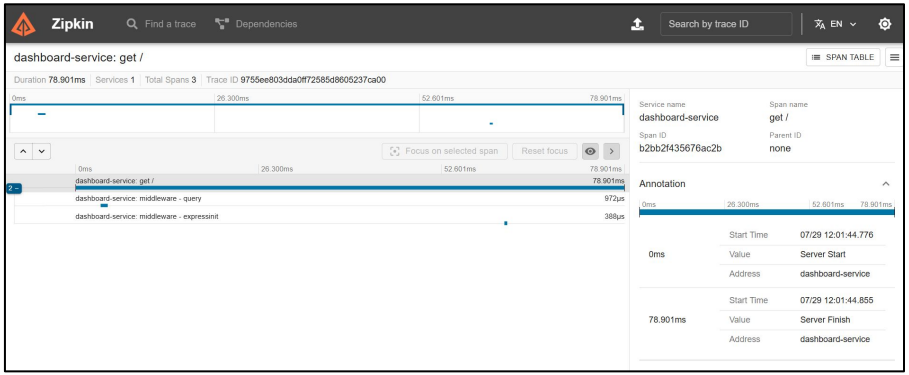
```
1  'use strict';
2
3  const { LogLevel } = require("@opentelemetry/core");
4  const { NodeTracerProvider } = require("@opentelemetry/node");
5  const { SimpleSpanProcessor } = require("@opentelemetry/tracing");
6  const { ZipkinExporter } = require("@opentelemetry/exporter-zipkin");
7
8
9  const provider = new NodeTracerProvider({
10   logLevel: LogLevel.ERROR
11 });
12
13 provider.register();
14
15 provider.addSpanProcessor(
16   new SimpleSpanProcessor(
17     new ZipkinExporter({
18       serviceName: "getting-started",
19     })
20   )
21 );
```

- Initialized **OpenTelemetry Tracer** in Node.js app
- Configured **Zipkin exporter** to collect and visualize trace data
- Registered tracing provider with ERROR-level logs for clean output
- SimpleSpanProcessor batches and sends trace spans to backend

OpenTelemetry

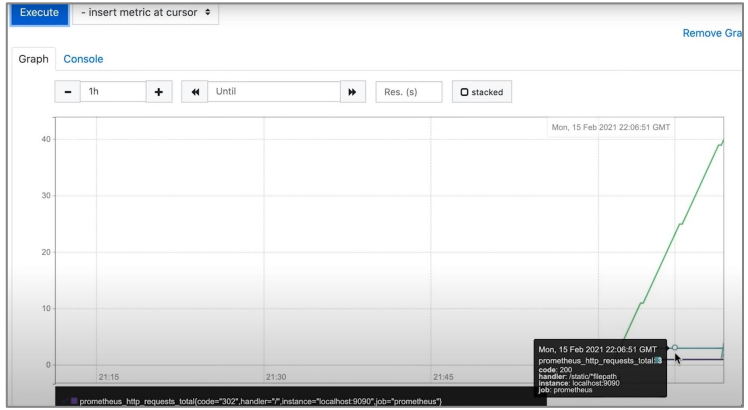
Kasadara

OUTPUT



Traces are found in the Zapkins

Monitor the model in prometheus



OpenTelemetry

Kasadara

LangSmith

LangSmith is a developer tool designed for tracing, debugging, and evaluating large language model (LLM) applications.

It provides detailed visibility into the internal steps of chains and agents — including prompts, tool calls, and intermediate outputs.

Built to work seamlessly with the LangChain framework, supporting both local and cloud-based models.

Enables faster iteration and better reliability for building complex AI applications by showing how the model "thinks."

Kasadara

LangSmith Setup

STEP 1 — Install Required Packages

```
PS D:\CODE\AI_Python\POC> pip install langchain langsmith transformers
```

STEP 2 — Import Modules

```
1 import os
2 from langchain.agents import initialize_agent
3 from langchain_community.agent_toolkits import load_tools
4 from langchain.llms import HuggingFacePipeline
```

STEP 3 — Set Environment Variables

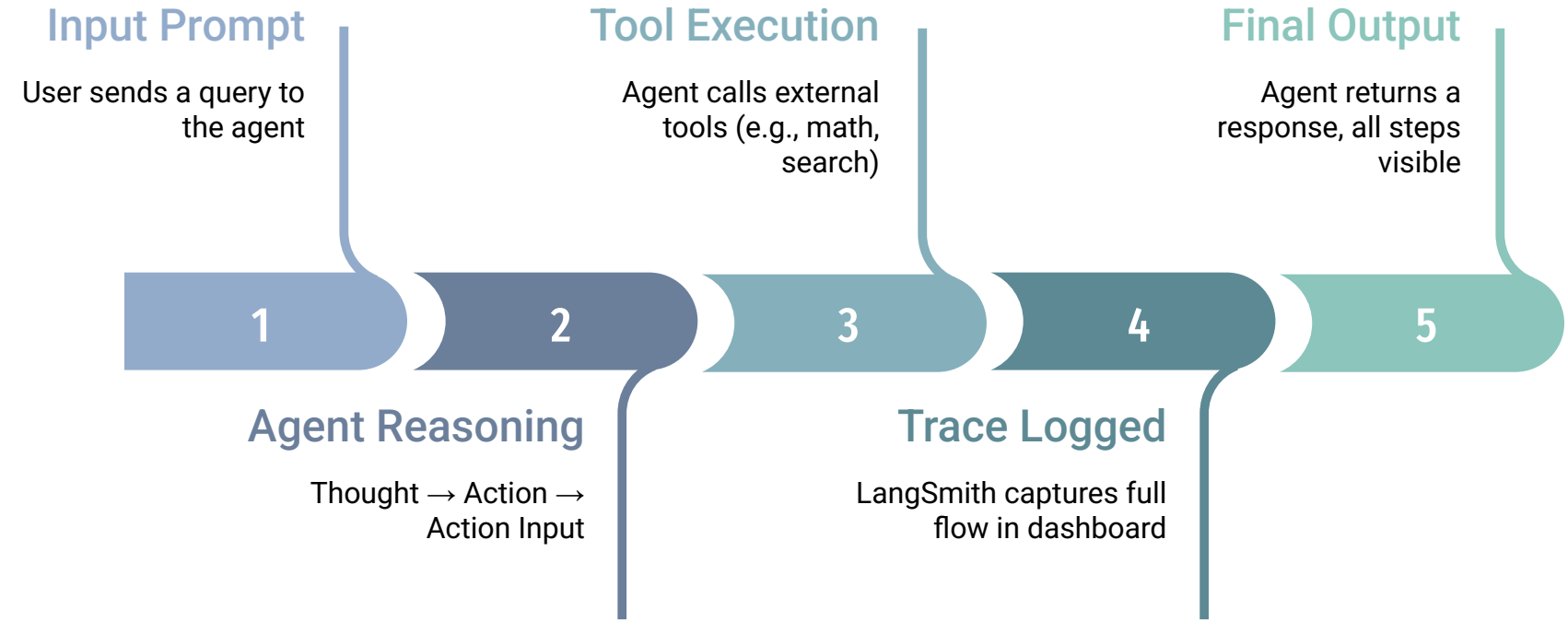
```
1 os.environ["LANGCHAIN_TRACING_V2"] = "true"
2 os.environ["LANGCHAIN_API_KEY"] = "your-api-key"
3 os.environ["LANGCHAIN_PROJECT"] = "langsmith-poc"
```

STEP 4 — Run the Agent

```
1 agent.run("PROMTPT")     agent
```

Kasadara

LangSmith Flow



Kasadara

IMPLEMENTATION

CODE

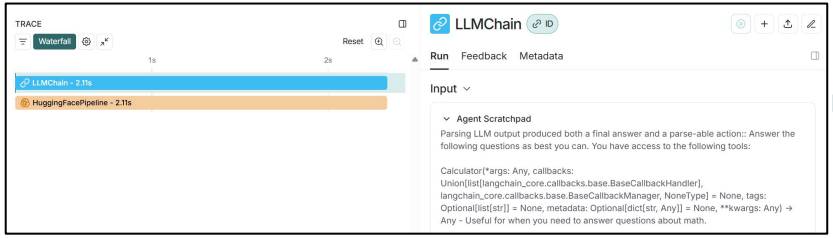
```
agent = initialize_agent(
            tools,
            llm,
            agent_type = strategy
        )

response = agent.invoke(PROMPT)

print(response)
```

tools	List of tools the agent can use (e.g., search, calculator)
llm	The LLM model (like OpenAI or HuggingFace pipeline)
agent_type	Strategy used by the agent

STRATEGIES

zero-shot-react-description - Most common; ReAct based on tool descriptions
react-docstore - Uses ReAct pattern with a docstore tool
self-ask-with-search - Asks follow-up questions and uses search tools to find answers
conversational-react-description - Like zero-shot but supports memory & chat history
chat-zero-shot-react-description - Optimized for chat models like OpenAI ChatGPT.

OUTPUT



Waterfall View
Shows execution time of each component (LLMChain + HuggingFacePipeline)
Helps identify **latency bottlenecks** in your chain

```
Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [Calculator]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question
```

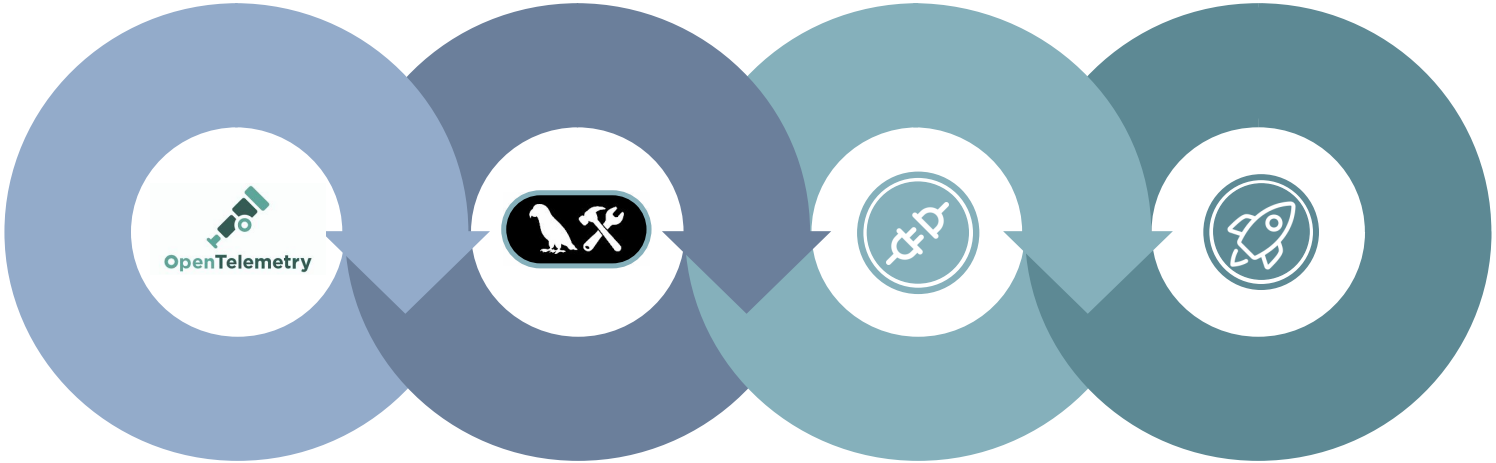
Agent Scratchpad + Prompt Format
Displays how LangChain uses structured prompts
Shows the format used by ReAct agents (Thought → Action → Input → Observation)

```
Begin!

Question: What is 2 + 2 ?
Thought: Hmm, let me think...
Action: Calculator
Action Input: 2 + 2
Observation: 4
```

Reasoning in Action
Real trace of the agent solving 2 + 2
Demonstrates LLM's step-by-step reasoning and tool use

Kasadara

CONCLUSION



LANGSMITH

Clear tracing of LLM reasoning and tool use

OPENTELEMETRY

Tracks system performance and errors

COMBINED

Full visibility from model to infrastructure

RESULT

Faster debugging, smarter optimization