Monitoreo y Optimizacion

en compiladores como servicio

Proyecto 02

Universidad Nacional Autónoma de México
Facultad de Ciencias

Compiladores

Proyecto 2

Guzmán Bucio Luis Antonio
Espinosa Roque Rebeca
Morales Martínez Edgar Jesús

14 de diciembre de 2025

En este proyecto, investigamos cómo las plataformas Compiler-as-a-Service (CaaS) pueden incorporar telemetría y aprendizaje continuo a partir de compilaciones reales. Mediante análisis teórico y una implementación práctica con OpenTelemetry, demostramos que la telemetría en CaaS no mejora los algoritmos de compilación clásicos (register allocation, constant folding, etc.), que son funciones deterministas, sino que optimiza la infraestructura del servicio en la nube. Se implementó un sistema CaaS funcional en Java que valida empíricamente esta distinción, recolectando métricas operacionales (latencia, throughput, tasa de error) sin modificar el proceso de compilación. Los resultados confirman que el valor de la telemetría radica en la observabilidad distribuida para decisiones de escalamiento y planificación de capacidad, no en el aprendizaje de patrones para optimización de código.

0.1. Telemetría y Aprendizaje Continuo en Compiladores como Servicio

0.1.1. Pregunta de Investigación

¿Cómo pueden las plataformas *Compiler-as-a-Service* incorporar telemetría y aprendizaje continuo a partir de compilaciones reales?

0.1.2. Fundamentos Conceptuales

Compilador como Servicio 1 ***Compiler-as-a-Service***

Un sistema distribuido que expone la funcionalidad de un compilador a través de una interfaz de programación de aplicaciones (API), permitiendo el análisis sintáctico, semántico y la generación de código como servicios remotos accesibles mediante protocolos estándar de red.

Uno de los mayores avances en el mundo del desarrollo de compiladores actual ha sido la llegada de los compiladores como servicio (*Compiler-as-a-Service*), siendo la API de Roslyn, que es el SDK de .NET Compiler Platform desarrollado por Microsoft [2], en el que nos basaremos principalmente en el desarrollo de este documento. Roslyn utiliza herramientas de análisis de código para mejorar la calidad del código y los generadores de código facilitando la construcción de aplicaciones, siendo esta la misión principal de la API de Roslyn, convirtiendo a los Compiladores en plataformas (APIs) que se pueden usar para tareas relacionadas con código en herramientas y aplicaciones [1].

Telemetría en Sistemas Distribuidos 2

La telemetría en la nube utiliza herramientas de software para grabar y analizar información acerca de la infraestructura de la tecnología de la información (TI) que de otra forma sería difícil de recolectar. Esta técnica le permite a los profesionales de la TI observar componentes y monitorear aplicaciones de una forma profunda, con métricas que rastrean el rendimiento, utilización, consumo de energía y más [4].

Algo que se complementa muy bien con la telemetría es la supervisión y mejora continua que algunas empresas de servicio en la nube como Google [9] y Alibaba [10] ofrecen ya integradas en sus plataformas proporcionando asesoría para seguimiento de progreso, definición de objetivos de rendimiento, supervisión continua de tus sistemas, entre otras. Por lo general estos sistemas ya traen una telemetría integrada que puede ir desde frameworks open source con estándares en la industria hasta una implementación propia.

0.1.3. Análisis Estático versus Telemetría en Tiempo de Ejecución

Distinción Fundamental 1

Sea $\mathcal{A} : \text{AST} \to \mathcal{P}(\text{Error})$ un analizador estático y $\mathcal{T} : \text{Execution} \to \mathcal{M}$ una función de telemetría que mapea ejecuciones al espacio de métricas $\mathcal{M}$. Entonces:

$$\text{dom}(\mathcal{A}) = \{\text{código fuente}\}$$
$$\text{dom}(\mathcal{T}) = \{\text{trazas de ejecución}\}$$

Aunque a simple vista pueda parecer que Roslyn utiliza telemetría para su optimización, en realidad lo que utiliza son análisis estáticos para comprender el código, es decir, no ejecutan el código ni proporcionan otras ventajas de prueba. Sin embargo, puede señalar prácticas que te llevan a errores, código difícil de mantener, o mal uso de estándares en la industria. La principal diferencia entre un analizador estático y la telemetría es que el primero sólo se basa en el código ya existente, de cierta forma, mientras que el segundo recolecta todos los valores que te pueda arrojar una ejecución para posteriormente poder analizarlos.

0.1.4. Alcance de la Telemetría en Optimización de Compiladores

Independencia de Optimizaciones Clásicas 2

Sea $\mathcal{O} = \{\text{RA}, \text{CSE}, \text{DCE}, \text{CF}\}$ el conjunto de optimizaciones clásicas de compiladores (Register Allocation, Common Subexpression Elimination, Dead Code Elimination, Constant Folding) [6]. Entonces para todo $o \in \mathcal{O}$:

$$o : \text{IR} \to \text{IR}$$

es una función determinista que no requiere información externa de telemetría para su correctitud o mejora.

La idea de incorporar telemetría en un CaaS en realidad no nos ayudaría a mejorar el compilador en sí, pues los algoritmos de optimización para Compiladores como Register Allocations, common-subexpression elimination, Dead-code elimination y Constant folding [6] no necesitan de más datos sobre otros código para lograr ser "más óptimos", sino sólo del propio algoritmo y del código que se quiere optimizar. Sin embargo, algo en lo que sí se podría incorporar la telemetría podría ser en la mejora del servicio en la nube, optimizando el rendimiento y rastreando errores de una manera más efectiva, asegurando al usuario una experiencia favorable.

Observabilidad 3

A este proceso de dar seguimiento a los datos también se le conoce como observabilidad. La observabilidad va más allá del monitoreo tradicional pues te genera conocimiento contextual sobre el comportamiento del sistema a través de tres pilares [8]:

1. **Tracing**: Rastreo de distintas solicitudes de flujo a través de múltiples servicios.
2. **Metrics**: Medición de la "salud del sistema", latencia y rendimiento.
3. **Logging**: Captura de información de eventos específicos para depurar y auditar.

0.1.5. OpenTelemetry: Estándar de Observabilidad

OpenTracing y OpenCensus nacieron como estándares para rastreo de sistemas distribuidos en la industria, un sistema distribuido rastrea y captura el ciclo de vida de una petición así como fluye a través de diferentes servicios en una Arquitectura de Microservicios, así cada petición genera un rastreo a múltiples operaciones individuales. En 2019 OpenTracing y OpenCensus se fusionaron para dar creación

a OpenTelemetry (OTel) que rápidamente se posicionó como el estándar de observabilidad principal, habilitando la opción en plataformas de que los desarrolladores puedan formatear su información para enviarla a una herramienta de observabilidad, para posteriormente generar métricas y visualizaciones a partir de los datos [7].

Ventajas de OpenTelemetry 1

Dentro de sus principales ventajas se encuentra:

- Colección unificada de telemetría que combina trazas, métricas, y registros en un solo framework
- Soporte a múltiples backends
- Trabajo a través de múltiples ambientes como Kubernetes y plataformas de la nube
- Extensibilidad y escalabilidad

Arquitectura de OpenTelemetry 4

La Arquitectura de OpenTelemetry se divide en múltiples componentes que ayudan a recolectar, procesar, y exportar datos telemétricos [11]. Sea $\mathcal{OT}$ el sistema OpenTelemetry, definimos sus componentes como la 6-tupla:

$$\mathcal{OT} = (\text{Spec}, \text{Instr}, \text{Coll}, \text{Exp}, \text{K8s}, \text{FaaS})$$

donde cada componente cumple con las siguientes especificaciones:

1. **Specification (Especificaciones)**: Describe los requisitos y expectativas entre lenguajes para todas las implementaciones. La especificación define los tipos de datos y operaciones para generar y recolectar trazas, métricas y datos de registro, también define los requerimientos de un lenguaje específico para la implementación de la API y finalmente define el protocolo OpenTelemetry y convenciones semánticas independientes del proveedor.
2. **Instrumentación (SDKs y APIs)**: Que significa que el código de los componentes del sistema deben de emitir señales como métricas, trazas y registros, de modo que usando OpenTelemetry puedes instrumentar tu código de dos formas principales: *Code Based Solutions* y *Zero Code Solutions*.
3. **Collector (Colector)**: Un *Vendor-agnostic proxy* (proxy independiente de proveedor) que intermedia el tráfico o la comunicación entre sistemas que permite recibir, procesar y exportar datos telemétricos.
4. **Exporters (Exportadores)**: Se encarga de enviar datos telemétricos al colector de OpenTelemetry para asegurarse que fue exportado correctamente.
5. **Kubernetes operator (Operador de Kubernetes)**: El operador de OpenTelemetry es una implementación del operador de Kubernetes. Este operador maneja el colector de OpenTelemetry y la auto-instrumentación de las cargas de trabajo usando OpenTelemetry.
6. **Function as a Service assets (Activos de funciones como servicio)**: OpenTelemetry soporta varios métodos de monitoreo de funciones como servicio que son proporcionados por diferentes proveedores de la nube, donde la documentación principal se encuentra alrededor de Microsoft Azure, Google Cloud Platform, y Amazon Web Services (Las funciones AWS también son conocidas como lambda).

0.1.6. Propuesta de Métricas para CaaS

Una vez entendido el funcionamiento de OpenTelemetry podemos empezar a visualizar cómo se podría implementar dentro de un CaaS. Creemos que algunas de las métricas que podrían valer la pena recolectar son:

Conjunto de Métricas Propuestas 5

Sea $\mathcal{M}_{\text{CaaS}}$ el espacio de métricas para un sistema Compiler-as-a-Service. Definimos los siguientes subconjuntos:

$$\begin{aligned}
\mathcal{M}_{\text{red}} &= \{\text{uptime}, \text{latencia}, \text{throughput}\} \\
\mathcal{M}_{\text{app}} &= \{\text{tiempo_respuesta}, \text{tasa_error}, \text{requests}\} \\
\mathcal{M}_{\text{servidor}} &= \{\text{instancias_total}, \text{instancias_activas}\} \\
\mathcal{M}_{\text{dep}} &= \{\text{disponibilidad}, \text{estado_servicio}\} \\
\mathcal{M}_{\text{host}} &= \{\text{mem}, \text{disco}, \text{CPU}\}
\end{aligned}$$

donde $\mathcal{M}_{\text{CaaS}} = \mathcal{M}_{\text{red}} \cup \mathcal{M}_{\text{app}} \cup \mathcal{M}_{\text{servidor}} \cup \mathcal{M}_{\text{dep}} \cup \mathcal{M}_{\text{host}}$.

1. **Métricas de rendimiento de la red**: Que incluyen tiempo de actividad, latencia, y productividad.
2. **Métricas de la aplicación**: Tiempos de respuesta, solicitudes y tasas de error.
3. **Métricas del grupo de servidores**: Total de instancias, número de instancias en ejecución.
4. **Métricas de dependencia externa**: Disponibilidad y estado del servicio.
5. **Métricas del Host**: Uso de memoria, disco y CPU.

Extensión de Métricas Específicas al Dominio 2

La recolección de estos datos telemétricos a nuestro parecer podría extenderse incluso un poco más. Por ejemplo:

- Medir lenguajes y versiones utilizadas para la compilación permitiéndole a la empresa saber en cuales compiladores enfocar sus esfuerzo inmediatos pues al ser más óptimos son los que le permitirán obtener una mayor ganancia
- Compilaciones promedio por usuario
- Distribución de carga
- Tiempos de espera en caso de haberlos
- Picos de compilación

entre otros que sabemos que podrían existir.

0.1.7. Implementación Demostrativa

Para validar los conceptos teóricos, se implementó un sistema CaaS funcional con OpenTelemetry en Java 17.

Arquitectura del Sistema Implementado 6

Sea CaaS nuestro sistema, definimos su arquitectura como la 3-tupla:

$$\text{CaaS} = (\text{API}, \text{Compiler}, \text{Telemetry})$$

donde:

- API : Request → Response es la interfaz REST
- Compiler : SourceCode → Result ∪ Error es el compilador de expresiones
- Telemetry : Execution → Metrics es el sistema de recolección

Componentes Principales

API REST. Tres endpoints: `POST /api/compilar`, `GET /api/metricas`, `GET /api/salud`. Formato de entrada:

```
{"expresion": "2 + 3 * 4", "lenguaje": "ARITMETICA"}
```

Compilador. Implementa análisis léxico, sintáctico y evaluación. Gramática BNF:

```
Expr   ::= Term (('+' | '-') Term)*
Term   ::= Factor (('*' | '/') Factor)*
Factor ::= Number | '(' Expr ')'
```

Telemetría. Integración completa de OpenTelemetry con los tres pilares:

Traces: Cada compilación genera un span con tiempo de inicio/fin, estado y excepciones.

Metrics: Se implementaron todas las métricas de $\mathcal{M}_{\text{CaaS}}$: `latenciaPromedioMs`, `throughput`, `totalRequests`, `tasaError`, `lenguajesUtilizados`, `totalTokensProcesados`, `erroresSintacticos`. Todas usando estructuras thread-safe (`AtomicLong`, `ConcurrentHashMap`).

Logs: Sistema estructurado con timestamps ISO-8601 para auditoría.

Resultados Observados

Ejemplo de métricas recolectadas:

```
{
  "latenciaPromedioMs": 8.5,
  "throughput": 2.77,
  "totalRequests": 100,
  "requestsExitosos": 87,
  "requestsFallidos": 13,
  "tasaError": 13.0,
  "lenguajesUtilizados": {"ARITMETICA": 100},
  "totalTokensProcesados": 523,
  "erroresSintacticos": 13
}
```

Validación Empírica

La implementación valida la hipótesis central: la telemetría en un CaaS no mejora los algoritmos de compilación (que son deterministas), pero proporciona valor significativo para optimizar el servicio. Los datos recolectados son directamente accionables para decisiones de infraestructura: escalamiento de nodos, identificación de cuellos de botella y planificación de capacidad.

0.1.8. Conclusión

Creemos que la telemetría bien integrada en un CaaS podría impulsar el rendimiento del servicio así como mejorar la calidad y desempeño a nivel usuario, dándole a la empresa datos sólidos sobre el uso de su sistema. Debido a la complejidad creciente de los Compiladores cada vez más el uso de equipos que adopten flujos de trabajo distribuidos y enfoques basados en datos se volverán indispensables, por lo que comprender cómo funcionan y se desenvuelven dentro de un entorno es de gran utilidad tanto para los usuarios como para las empresas.

Bibliografía

[1] A. Sánchez, "Introduction to Roslyn: The Compiler as a Service," *ResearchGate*, Nov. 2024. [Online]. Available: https://www.researchgate.net/publication/390306032_Introduction_to_Roslyn_The_Compiler_as_a_Service

[2] Microsoft, "Roslyn SDK for C#," *Microsoft Learn*, 2025. [Online]. Available: https://learn.microsoft.com/es-es/dotnet/csharp/roslyn-sdk/

[3] A. Hejlsberg, "Compilers: What Every Programmer Should Know About Compiler Optimizations," *MSDN Magazine*, Microsoft, Feb. 2015. [Online]. Available: https://learn.microsoft.com/es-es/archive/msdn-magazine/2015/february/compilers-what-every-programmer-should-know-about-compiler-optimizations

[4] Intel Corporation, "Telemetry in Cloud Computing," *Intel*, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/cloud-computing/telemetry.html

[5] S. Podduturi, "OpenTelemetry – A Unified Approach to Observability in Microservices Architectures," *International Journal of Innovative Research in Computer Technology (IJIRCT)*, vol. 9, no. 6, pp. 1–16, Nov. 2023. doi: 10.5281/zenodo.15087101. [Online]. Available: https://www.ijirct.org/viewPaper.php?paperId=2503075

[6] W. Appel, *Modern Compiler Implementation in Java*, 2nd ed. Cambridge, U.K.: Cambridge University Press, 2002.

[7] M. E. Gortney, T. Cerny, and A. S. Abdelfattah, "Visualizing Microservice Architecture in the Dynamic Perspective," *IEEE Access*, vol. 10, pp. 173681–173709, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9944666/

[8] IBM, "The Three Pillars of Observability," *IBM Think Insights*, 2025. [Online]. Available: https://www.ibm.com/mx-es/think/insights/observability-pillars

[9] Google Cloud, "Continuously monitor and improve performance," *Google Cloud Architecture Framework*, 2025. [Online]. Available: https://docs.cloud.google.com/architecture/framework/performance-optimization/continuously-monitor-and-improve-performance?hl=es-419

[10] Alibaba Cloud, "Observability: The Three Pillars Explained," *Alibaba Cloud Blog*, 2025. [Online]. Available: https://www.alibabacloud.com/blog/602160

[11] OpenTelemetry, "Components" *OpenTelemetry Documentation*, 2025. [Online]. Available: https://opentelemetry.io/docs/concepts/components/#collector