Pregunta 2: What I Wish I Had Known Before Scaling Uber to 1000 Services.

La charla comienza indicando la increíble tasa de crecimiento de Uber citando de ejemplo 2 ciudades de China: Beijing y Chengdu, desde inicios hasta fin de año. El mismo patrón sucede en más de 400 ciudades y en más 70 países alrededor del mundo.

Esta rápida tasa de crecimiento conlleva a la implementación de microservicios, sin embargo, el momento en que es más probable que los sistemas se rompan es cuando los cambias, por lo que nos explica las ventajas de los microservicios.

- **Ventajas:**
  - El movimiento y la liberación de forma independiente de cada equipo de trabajo en lugar de una sola aplicación monolítica.
  - Conocer el tiempo de operatividad de tus servicios y tu propio código.
  - Usar la mejor herramienta para el trabajo.

El costo de hacer estas grandes implementaciones de micro servicios son por ejemplo el que ahora tienes un sistema distribuido que es mucho más difícil de trabajarlo que una única parte de software, el que todo ahora es un RPC y el peligro constante de que tu servicio se "rompa"

También existen menores costos como que todo es una compensación, por ejemplo tú puedes elegir el construir un nuevo servicio en vez de arreglar uno que está dañado, construir software alrededor de problemas y nunca limpiar los software anteriores, poder negociar complejidad por política creando más software, poder mantener tus prejuicios, por ejemplo escribir software en el lenguaje que tu creas apropiado

Uber empezó con envíos escritos en Node.js y Python, pero ahora el equipo está cambiando a Go y eventualmente a Java, lo que significa que el equipo está escribiendo en un total de 4 idiomas y aun así se puede seguir comunicando, esto gracias a los microservicios.

Aun así siempre hay costos para operar de esta manera siendo estos:

- **En el lenguaje:**
  - Código difícil de compartir.
  - Es difícil de moverse entre los equipos.
  - Tener muchos lenguajes puede fraccionar la cultura.
- **RPC:**
  - HTTP/REST se complica por problemas de interpretación.
  - JSON necesita un esquema.
  - Servidores no son buscadores, es mejor tratar como llamada de función en vez de solicitud WEB.
- **Repositorios:**
  - Muchos repositorios son buenos, para manejar módulos más pequeños (Uber en 2016 tenía más de 7000 repos y más adelante ese mismo año 8000).
  - Un repositorio es bueno: porque trae beneficios como hacer cambios transversales si quieres hacer cambios en 2 módulos al mismo tiempo.
  - Muchos repositorios son malos: porque puede ser muy estresante construir tu sistema.

- Un repositorio es malo: eventualmente va a ser tan grande que tú no vas a poder construir tu software.

- **Operacional:**
  - ¿Qué hacer cuando las cosas se rompen? Hay mucha dependencia, quizá algunos equipo están bloqueado por ti porque no están listos para lanzar tu servicio pero tienen que ponerle una solución
  - Otros equipos pueden liberar tu servicio? Depende de la situación pero usualmente vas a estar bloqueado por otro equipo lanzando su servicio por lo que se necesita coordinación
  - Entender un servicio en el largo contexto: trabajar como un conjunto

- **Performance:**
  - Depende de las herramientas del lenguaje: las herramientas son todas diferentes, todos quieren un dashboard pero si estos no se generan automáticamente, los equipos van a terminar creando sus propios dashboards, todos los servicios cuando son creados tienen el dashboard estándar con el mismo set de cosas útiles para el servicio
  - No importa hasta que importa: Suele no considerarse el performance, pero va a llegar el día en que tengas un problema que si no lo soluciones se va a agravar, por lo que siempre tienes que saber cómo se encuentra tu performance
- **Fanout:**
  - Latencia general >= latencia más lenta: latencia de una solicitud está dominada por la latencia más lenta del fan-out.
  - Si se tiene un servicio que es bastante rápido el 99% de las veces pero el 1% del tiempo toma un segundo, entonces tienes un 1% de tus usuarios teniendo el retraso.
  - La mejor manera de lidiar con los problemas de fan-out es tener tipos de rastreos
- **Tracing:**
  - Muchas formas de obtener rastreos, puede ser sencillo como pasar un identificador a través de un request.
  - La mejor manera de entender fanout es mediante el tracing.
  - La sobrecarga del tracing en realidad cambia los resultados.
  - Implementar el comportamiento del tracing requiere la propagación del contexto de lenguaje cruzado.
- **Logging:**
  - Necesita consistencia y estructura en el logging, todos tratan de hacer login en diferentes incompatibles maneras y se hace necesario la consistencia y estructura.
  - Múltiples lenguajes lo complican: cuando hay problemas, iniciar sesión puede hacer que esos problemas empeoren e inunden el sistema de logging.
  - Los registros se pueden analizar y estructurar mediante sistemas de logging como Splunk, Elk o Hadoop y luego ser utilizado para el consumo humano.
- **Load Testing:**
  - Necesidad de realizar test de la producción sin romper las métricas.
  - Ejecutar los test en todos los servicios preferiblemente todo el tiempo incluso cuando hay un exceso de capacidad.

- Todos los sistemas necesitan manejar el tráfico de prueba.

- **Failure testing:**
  - Los equipos odian el failure testing porque están matando el servicio que ellos crean con el fin de buscar errores.
  - Esto no es excusa para dejar de hacerlo, porque de todas maneras se tienen que realizar.
- **Open Source:**
  - Construir o comprar es una transacción difícil.
  - Siempre hay alguien que puede hacerlo mejor y más barato, por lo que causa desmoralización en la gente.
- **Políticas:**
  - Servicios permiten a las personas jugar con las políticas: sucede cuando la compañía > equipo> uno mismo, es decir estarías violando esta propiedad cuando te pones a ti mismo por encima del equipo, o el equipo se pone por encima de la compañía.

- **Tradeoffs:**
  - Optimizar la forma en que se observa los tradeoffs ayuda a trabajar mejor con los microservicios y sus deficiencias.