

Arquitectura de Software

Atributos de calidad

Escribirlos, medirlos y priorizarlos

Gerson Garrido

Semana 2

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A

1. Qué hace y qué tan bien lo hace
2. El problema de las palabras
3. El escenario de seis partes
4. Los atributos que más veremos

Sesión B

5. Percentiles
6. Medir sin engañarse
7. Disponibilidad
8. Utility tree y trade-offs

Lecturas

A: *Software Architecture in Practice*, cap. 3; Barbacci y otros, *Quality Attributes* (SEI, 1995), secciones 1 y 2.

B: *Designing Data-Intensive Applications* (2.ª ed.), cap. 2, *Describing Performance* y *Reliability and Fault Tolerance*; *Software Architecture in Practice*, cap. 19, el utility tree; *Site Reliability Engineering*, cap. 4.

De la lista de la semana 1 a algo que se comprueba

Lo que salió del laboratorio

“El catálogo debe aguantar la apertura de una venta.” “Debe ser escalable.” “Debe ser seguro.”

- Si no se puede decir cuándo se incumple, todavía no es un requisito.
- **Escribir:** un escenario de seis partes que termina en una medida.
- **Medir:** percentiles y disponibilidad, sin engañarse.
- **Priorizar:** un utility tree; después, elegir qué se sacrifica.

Todo lo que se mide hoy es Tinkuy v0, tal como está.

SECCIÓN 01

Qué hace y qué tan bien lo hace

01

Tres cosas que se le piden a un sistema

	Pregunta	En Tinkuy	Se comprueba con
Requisito funcional	¿Qué hace?	Comprar de 1 a 6 asientos de un evento	Una prueba de entrada y salida
Atributo de calidad *(quality attribute)*	¿Qué tan bien lo hace?	La compra responde en menos de 1 s con 200 compradores a la vez	Una medición bajo un estímulo
Restricción *(constraint)*	¿Qué ya está decidido?	Todo corre en el contenedor, sin paquetes nuevos	Se revisa; no se mide

La funcionalidad no determina la arquitectura

Vender una entrada se puede construir con:

- un proceso y una transacción en PostgreSQL (Tinkuy v0);
- tres servicios y una cola;
- una hoja de cálculo y una persona al teléfono.

Las tres venden una entrada. Lo que descarta a la hoja de cálculo es qué tan bien lo hace.

Idea clave

La funcionalidad dice qué tiene que existir. Los atributos de calidad dicen cómo tiene que estar organizado.

En Tinkuy

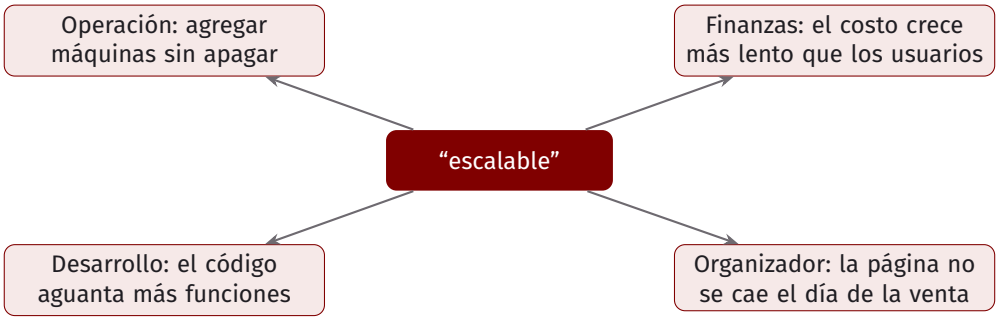
`make check` pasa en v0. Cada entrega trae un atributo que el diseño no cumple.

SECCIÓN 02

El problema de las palabras

02

Una palabra, muchos significados



“Tinkuy debe ser escalable” no descarta ningún diseño.

Un hecho, muchos nombres

build/tinkuy.log, durante make load-sale

```
POST /orders 500 error: sorry, too many clients already
```

200 compradores a la vez, 100 errores.
¿Qué atributo falló?

- ¿Rendimiento? No atendió la carga.
- ¿Disponibilidad? Para esos 100, estuvo caído.
- ¿Escalabilidad? No hubo cómo agregar capacidad.

La salida

Dejar de discutir el nombre y **escribir un escenario**: qué llega, a qué parte, en qué condiciones, qué debe pasar y con qué número se comprueba.

SECCIÓN 03

El escenario de seis partes

03

Las seis partes



- 1 a 3: la situación. 4: dónde ocurre. 5 y 6: qué se espera.
- Es una prueba escrita en prosa: preparación, lo que se observa y la aserción.

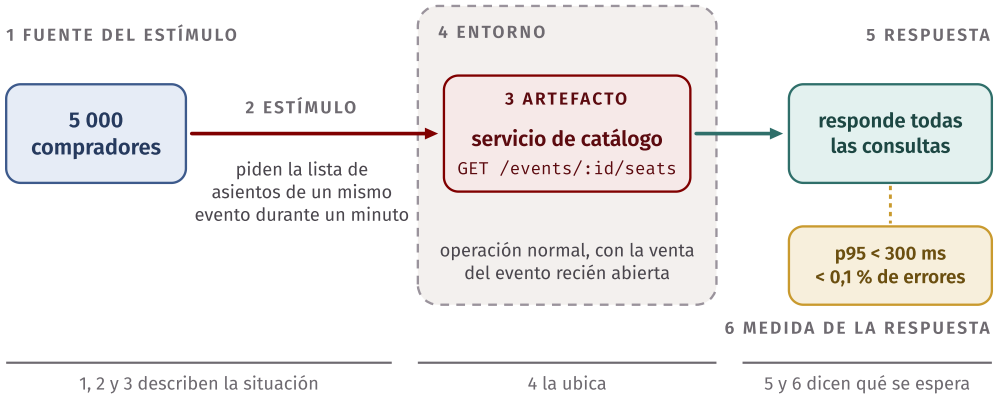
> **Idea clave**
> Sin medida no es un escenario.

Un escenario de Tinkuy

Parte	**“El catálogo debe aguantar la apertura de una venta”**
Fuente del estímulo	5 000 compradores
Estímulo	Piden la lista de asientos de un mismo evento durante un minuto
Artefacto	El servicio de catálogo (`GET /events/:id/seats`)
Entorno	Operación normal, con la venta del evento recién abierta
Respuesta	El sistema responde todas las consultas
Medida de la respuesta	Latencia p95 menor que 300 ms y menos de 0,1 % de errores

Ahora se puede preguntar: ¿por qué 300 ms?, ¿de dónde salen los 5 000?
Esas preguntas se le hacen al stakeholder, no al código.

El escenario, dibujado



Escenario general y escenario concreto

General

Para un atributo, los valores posibles de cada parte. Un menú, sin sistema ni números.

Rendimiento: llegan eventos (periódicos, esporádicos, impredecibles), en operación normal, pico o sobrecarga; se mide latencia, throughput, tasa de errores.

Concreto

El de un sistema y una situación, con números. Sirve para diseñar y para probar.

5 000 compradores, un minuto, `GET /events/:id/seats`, p95 menor que 300 ms.

El general sirve para no olvidar partes (¿y en sobrecarga?) y para entrevistar: no “¿qué rendimiento necesita?”, sino “¿cuánta gente llega a la vez el día de una venta grande?”.

El template

scenarios/tinkuy-seat-list.md

```
## La lista de asientos cuando se abre una venta

- **Atributo de calidad:** rendimiento
- **Fuente del estímulo:** 5 000 compradores
- **Estímulo:** piden la lista de asientos de un mismo evento durante un minuto
- **Artefacto:** el servicio de catálogo (‘GET /events/:id/seats‘)
- **Entorno:** operación normal, con la venta del evento recién abierta
- **Respuesta:** el sistema responde todas las consultas
- **Medida de la respuesta:** latencia p95 menor que 300 ms y menos de 0,1 % de
  errores, medidos desde quien llama durante ese minuto
```

En el proyecto: `docs/scenarios.md`, con este formato. Crece con cada entrega.

El revisor

```
node scenario-check.ts scenarios/tinkuy-seat-list.md scenarios/qhatu.md

scenarios/tinkuy-seat-list.md
  ✓ La lista de asientos cuando se abre una venta (línea 6)
scenarios/qhatu.md
  ✓ E1. Búsqueda de productos en la hora punta (línea 8)
  ✓ E2. Posición del repartidor (línea 18)
  ✓ E3. El procesador de pagos deja de responder (línea 28)
  ✓ E4. Se cae una instancia del servidor (línea 38)
  ✓ E5. Abrir el servicio en una ciudad nueva (línea 48)
  ✓ E6. Un comprador pide datos de otro (línea 58)
```

- Exige las seis partes y un número en la medida.
- Revisa la forma, no el fondo: no sabe si 300 ms es lo que el stakeholder necesita.
- La prueba que vale: otra pareja comprueba la medida sin preguntarles nada.

Error común: sin medida, o con una que nadie puede comprobar

```
make errors

errors/vague-scenarios.md
  ✗ Sin medida: "responde rápido" (línea 6)
      Medida de la respuesta: no tiene ningún número: ¿cómo se comprueba?
      Medida de la respuesta: "rápido" no se puede comprobar: escriban el número
  ✗ Con una medida que nadie puede comprobar: "nunca falla" (línea 16)
      Medida de la respuesta: "nunca" no se puede comprobar: escriban el número
  ✗ Sin entorno (línea 26)
      Entorno: falta
scenario-check terminó con código 1
```

- “Rápido”: quien lo escribió siempre puede decir que 4 segundos le parecen rápidos.
- “Nunca falla”, “100 %”: después de un año sin fallas todavía no se sabe si se cumple.

Error común: un escenario sin entorno

“La búsqueda responde con p95 menor que 400 ms”

¿Con cuánta carga? ¿Con cuántos datos? ¿En qué estado está el sistema?

- Casi cualquier sistema responde en 400 ms con un usuario y la base de datos vacía.
- El entorno distingue la demostración del día de la venta: *en hora punta*, *con el catálogo cargado*, *con una réplica caída*, *durante un deploy*.
- En disponibilidad es la mitad del requisito: *“con la pasarela respondiendo en 5 segundos”*.

El estímulo que se mide

El escenario pide 5 000 compradores; `make load-browse` genera 50 usuarios en una laptop. Es otra situación: escriban el estímulo que de verdad generaron.

Los atributos que más veremos, por su estímulo

Atributo	Estímulo típico	La medida suele ser
Disponibilidad	Una falla	Fracción atendida; tiempo para recuperarse
Rendimiento	Llegan eventos	Latencia por percentil, throughput
Modificabilidad	Llega un cambio	Esfuerzo, módulos tocados
Capacidad de prueba	Hay que probar una parte	Tiempo para preparar y correr la prueba
Capacidad de despliegue	Hay una versión nueva	Tiempo hasta producción; para volver atrás
Seguridad	Un ataque	Datos expuestos, tiempo hasta detectarlo
Escalabilidad	La carga crece	Recursos para mantener otra medida

Ninguna lista es completa (ISO/IEC 25010 propone otra, con nueve características).
Los nombres importan menos que el escenario.

Tácticas: una primera mención

> **Táctica** *(tactic)*
> Una decisión de diseño que influye sobre la respuesta del sistema a un estímulo.

- Para la lista de asientos: guardar la respuesta en una cache, mantener abiertas las conexiones, limitar cuántos pedidos se atienden a la vez.
- Cada una mueve la medida, y cada una cuesta algo en otro atributo.
- El orden: primero el escenario, después la táctica. Quien empieza por “pongamos una cache” no sabe si la necesitaba ni si funcionó.

El catálogo de tácticas por atributo es la semana 6.

SECCIÓN 04

Percentiles

04

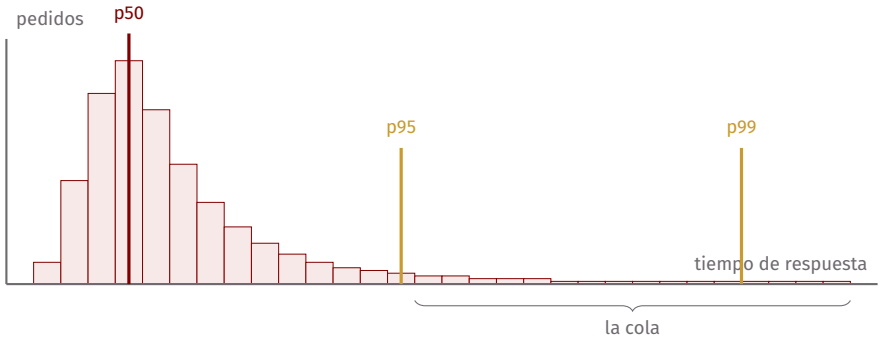
Latencia, tiempo de respuesta y throughput

- **Tiempo de respuesta:** lo que ve quien llama, de enviar a recibir. Red, cola y trabajo.
- **Latencia:** en sentido estricto, la espera. En la práctica se usa como sinónimo; aquí también.
- **Throughput:** trabajo completado por unidad de tiempo.

k6	Qué mide
`http_req_duration`	Tiempo de respuesta
`http_reqs`	Pedidos, y por segundo
`http_req_failed`	Fracción que falló

Un escenario de rendimiento fija los dos: el estímulo dice cuántos pedidos llegan; la medida, cuánto puede tardar cada uno.

El tiempo de respuesta es una distribución



Esquema, no una medición.

El mismo pedido tarda distinto cada vez. El problema es resumir la distribución sin esconder la cola.

Percentil: definición y cálculo

Percentil p, por rango más cercano *(nearest-rank)*

Se ordenan las n mediciones de menor a mayor. El percentil p es la que ocupa la posición $\lceil p/100 \times n \rceil$, contando desde 1.

stats.ts

```ts
export function percentile(values: readonly number[], p: number): number {
  // (se rechazan las listas vacías y los p fuera de (0, 100])
  const sorted = [...values].sort((a, b) => a - b);
  const rank = Math.ceil((p / 100) * sorted.length);
  return sorted[rank - 1]!;
}
```

p50 es la mediana. p95 deja arriba a 1 de cada 20 pedidos; p99, a 1 de cada 100.

Un ejemplo a mano

Cinco mediciones: 15, 20, 35, 40 y 50 ms.

	$p/100 \times n$	**Posición**	**Valor**
p30	1,5	2	20 ms
p40	2,0	2	20 ms
p50	2,5	3	35 ms
p100	5,0	5	50 ms

- El percentil siempre es una de las mediciones.
- Con 20 mediciones, el p99 es el máximo: los percentiles altos necesitan muchas muestras.
- Habla de pedidos, no de usuarios.
- k6 interpola: con 10, 20, …, 100 ms reporta mediana de 55 ms y p95 de 95,5 ms; el rango más cercano da 50 y 100. Digan siempre con qué método se calculó.

Por qué no el promedio

98 respuestas de 50 ms y 2 de 5 s:

```
node calc.ts percentiles data/latencies.txt

n = 100
promedio = 149.0 ms
p50 = 50 ms   p95 = 50 ms   p99 = 5000 ms   máx = 5000 ms
```

- Nadie esperó 149 ms. El promedio no describe a ningún usuario.
- Pocos valores extremos lo arrastran, y mezcla poblaciones.
- Sirve para capacidad y costo; no para decir qué le pasó a la gente.

Los 100 pedidos, uno por punto



Tinkuy: dos poblaciones en la venta

make load-sale (200 compradores, 30 s; laptop con Apple Silicon, en Docker)

```
orders_error..................: 100    3.165525/s
orders_paid...................: 1427   45.172035/s
orders_seat_taken.............: 4242   134.28155/s

http_req_duration.............: avg=58.34ms min=3.8ms   med=6.01ms p(90)=210.3ms  p(95)=212.21ms p(99)=270.45ms max=404.59ms
http_req_failed...............: 1.73 %  100 out of 5770
http_reqs.....................: 5770   182.650765/s
```

- Mediana de 6 ms, p90 de 210 ms: no es una cola, es un salto.
- 4 242 rechazos (409) que responden en unos 6 ms; 1 427 compras que pasan por la pasarela (80 ms) y el correo (120 ms).
- El promedio, 58 ms, no es de nadie. Y la mediana describe al que no compró.

Si una métrica mezcla trabajos distintos, se separa antes de resumirla.

Por qué importan los percentiles altos

- Los pedidos lentos no se reparten al azar: suelen ser de quienes tienen más datos, que son quienes más usan el sistema.
- Una pantalla hace muchas llamadas y espera a la más lenta.

El 1 % del servicio

Para una pantalla de 100 llamadas, lo raro del servicio es lo normal.

Probabilidad de que al menos una de n llamadas independientes supere el p99: $1 - 0{,}99^{n}$

Llamadas	Probabilidad
1	1 %
10	9,6 %
100	63,4 %

Los percentiles no se promedian

Servidor rápido
900 pedidos de 20 ms
p95 = 20 ms

Servidor lento
100 pedidos de 800 ms
p95 = 800 ms

- Promedio de los dos p95: 410 ms. Ningún pedido tardó eso.
- p95 real: la posición 950 de 1 000. Es de 800 ms.
- Un percentil es una posición en la lista ordenada: hace falta la lista completa.
- Sí se combinan: las mediciones crudas o los histogramas.

SECCIÓN 05

Medir sin engañarse

05

Medir Tinkuy con k6

make load-browse (hasta 50 usuarios virtuales, 30 s; laptop con Apple Silicon, en Docker)

```
duration_event_detail..........: avg=8.56ms  min=6.47ms  med=7.96ms  p(90)=10.74ms p(95)=12.06ms p(99)=16.73ms max=27.97ms
duration_event_seats...........: avg=11.79ms min=8.73ms  med=10.94ms p(90)=14.81ms p(95)=17.07ms p(99)=24.31ms max=34.25ms
duration_events_list...........: avg=61.47ms min=48.65ms med=57.66ms p(90)=76.05ms p(95)=82.72ms p(99)=111.8ms max=126.28ms

http_req_duration..............: avg=27.27ms min=6.47ms  med=11.4ms  p(90)=64.56ms p(95)=71.18ms p(99)=91.85ms max=126.28ms
http_req_failed................: 0.00%  0 out of 2205
http_reqs......................: 2205   71.351582/s
```

- 71 pedidos por segundo: lo que el script generó, no el máximo de Tinkuy.
- La mediana (11 ms) y el p95 (71 ms) hablan de endpoints distintos: el p95 global es la lista de eventos.
- Lo que el resumen no dice y hay que anotar: la máquina y los datos.

Cuánto varía una medición

Tres corridas seguidas de `make load-browse`, sin cambiar nada (`http_req_duration`):

Corrida	Promedio	Mediana	p95	p99	Máximo
1	27,27 ms	11,4 ms	71,18 ms	91,85 ms	126,28 ms
2	39,21 ms	12,96 ms	105,05 ms	352,83 ms	817,36 ms
3	30,75 ms	12,52 ms	81,78 ms	107,19 ms	249,19 ms

- El p95 cambió casi 50 % y el p99 casi se cuadruplicó. La mediana, casi nada.
- Una corrida no alcanza. Un cambio cuenta cuando los rangos de antes y de después no se tocan.
- Cuanto más alto el percentil, más varía: hacen falta corridas más largas.

Métricas por endpoint

make endpoints (load/by-endpoint.js: 20 usuarios virtuales, 20 s; misma laptop)

```
http_req_duration..................: avg=29.02ms med=11.18ms p(95)=113.29ms p(99)=185.1ms  max=199.38ms
  { name:GET /events }.............: avg=99.84ms med=84.7ms  p(95)=185.16ms p(99)=195.74ms max=199.38ms
  { name:GET /events/:id }.........: avg=11.42ms med=9.9ms   p(95)=21.97ms  p(99)=29.89ms  max=33.11ms
  { name:GET /events/:id/seats }...: avg=15.22ms med=14.37ms p(95)=23.7ms   p(99)=29.54ms  max=31.14ms
  { name:GET /orders/:id }.........: avg=10.06ms med=9.43ms  p(95)=15.7ms   p(99)=17.7ms   max=25.94ms
  { name:GET /reports/sales }......: avg=8.58ms  med=8.12ms  p(95)=13.19ms  p(99)=16.74ms  max=20.07ms
http_req_failed....................: 0.00%  0 out of 1800
```

- Cada pedido lleva una etiqueta `name`; la métrica se filtra por ella.
- `GET /events`: entre seis y diez veces la mediana de los otros. Esa comparación sí viaja a otra máquina.
- Devuelve 24 eventos; el de asientos devuelve hasta 2 000 y es rápido. ¿Por qué? Laboratorio, paso 4.

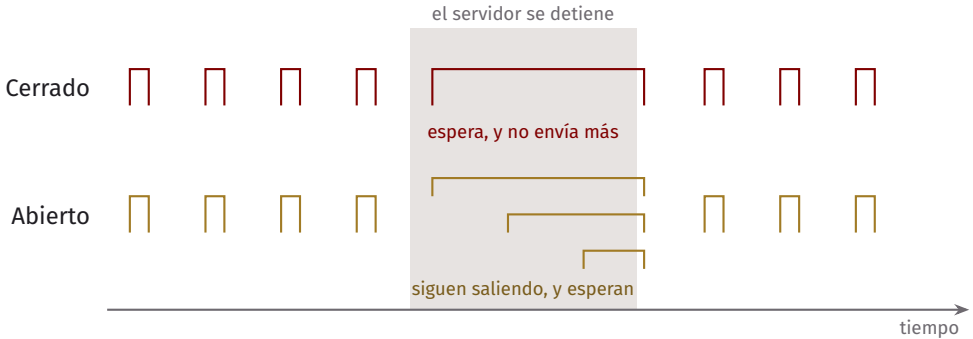
Tres cuidados antes del más sutil

- **Carga realista.** La mezcla de pedidos, los datos cargados y las pausas se parecen al estímulo del escenario. Después de `make load-sale` la base de datos es otra: `make db` antes de repetir.
- **Calentamiento.** Los primeros pedidos tras arrancar suelen ser más lentos. Se descartan, o se usa una rampa… salvo que el escenario sea “el sistema acaba de reiniciarse”.
- **Desde quien llama.** El servidor que se mide a sí mismo no ve la cola ni la red. k6 mide desde afuera.

Error común

Comparar números de dos scripts distintos. Una latencia solo se compara con otra del mismo script, los mismos datos y la misma máquina.

Dos modelos de generador de carga



- **Cerrado** (`constant-vus`): enviar, esperar la respuesta, descansar, enviar.
- **Abierto** (`constant-arrival-rate`): los pedidos salen a un ritmo fijo, responda o no el servidor.

Un servidor que se detiene

omission/stall-server.ts

Responde en 2 ms, salvo durante una ventana de 1 s que se repite cada 10 s: lo que llega en la ventana espera hasta que termine.

Antes de medir: ¿qué debería reportar una medición honesta?

- Está detenido el 10 % del tiempo.
- Si la gente llega a ritmo parejo, el 10 % de los pedidos cae en una ventana y espera entre 0 y 1 s.
- p90 en el borde; p95 cerca de 500 ms; p99 cerca de 900 ms.

`make omission`: el mismo servidor, el mismo umbral `p(95)<300`, los dos modelos.

El mismo servidor, dos conclusiones

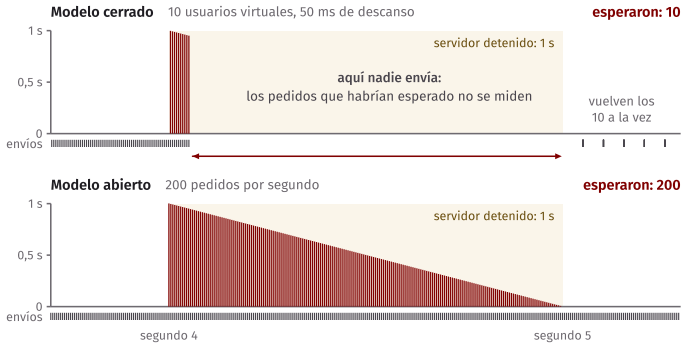
make omission (laptop con Apple Silicon, en Docker)

```
== Modelo cerrado: 10 usuarios virtuales (constant-vus)
    ✓ 'p(95)<300' p(95)=5.71ms
    http_req_duration..............: avg=9.84ms  med=3.57ms  p(90)=5.23ms  p(95)=5.71ms  p(99)=7.81ms  max=1s
    http_reqs......................: 4811   160.250969/s
k6 terminó con código 0
== Modelo abierto: 200 pedidos por segundo (constant-arrival-rate)
    ✗ 'p(95)<300' p(95)=502.9ms
    http_req_duration..............: avg=53.57ms med=3.43ms p(90)=14.72ms p(95)=502.9ms  p(99)=901.47ms max=1s
    http_reqs......................: 6002   200.032801/s
k6 terminó con código 99
```

- Mediana y máximo coinciden. En el medio: p95 de 5,71 ms contra 502,9 ms.
- Cerrado: en cada detención queda atrapado un pedido por usuario. A lo sumo 30 lentos entre 4 811: 0,6 %.
- Abierto: salieron los 200 por segundo que tocaban. 600 esperaron: el 10 %.

Una detención, vista con los dos modelos

CADA MARCA ES UN ENVÍO; CADA BARRA, LO QUE ESE PEDIDO ESPERÓ LA RESPUESTA



El mismo servidor, dos conclusiones: el modelo cerrado no envió los pedidos que habrían esperado.

MAKE OMISSION: 30 S, 3 DETENCIONES	esperaron	del total	p95 medido con k6
modelo cerrado	3 × 10 = 30 pedidos	**0,6 %**	**5,71 ms**
modelo abierto	3 × 200 = 600 pedidos	**10 %**	**502,9 ms**

Omisión coordinada

Omisión coordinada *(coordinated omission)*

El generador espera cada respuesta antes de enviar el siguiente pedido, y por eso deja de medir justo cuando el sistema se pone lento.

- ¿Cuál modelo es el correcto? **Depende de a quién representa la carga.**
- Abierto: gente que llega por su cuenta. La apertura de una venta.
- Cerrado: una población fija que de verdad espera. Diez cajeros, cuatro consumidores de una cola.
- El abierto cuesta: hay que elegir la tasa y reservar usuarios virtuales. Con 20 en vez de 400, k6 descartó 544 pedidos (`dropped_iterations`) y el p95 volvió a 4,05 ms.

Los scripts de Tinkuy usan el modelo cerrado: sirven para comparar dos diseños, sabiendo esto.

Umbrales: la medida convertida en prueba

el escenario de la lista de asientos, en k6

```
export const options = {
  thresholds: {
    'http_req_duration{name:GET /events/:id/seats}': ['p(95)<300'],
    'http_req_failed{name:GET /events/:id/seats}': ['rate<0.001'],
  },
};
```

- Si no se cumple, k6 termina con código distinto de cero (99): un `make` falla.
- El filtro por etiqueta juzga al endpoint del escenario, no a la mezcla.
- Un umbral vale lo que vale la carga: el mismo servidor pasó con el modelo cerrado y falló con el abierto.

Dos errores comunes al medir

Fijar el umbral después de medir

“Salió 420 ms, entonces el requisito es 500.”

Así siempre se cumple y no dice nada. El umbral sale de lo que necesita el stakeholder. Medir primero sirve para saber qué tan lejos se está.

La laptop no es el sistema

“Tinkuy responde la lista de eventos en 85 ms.”

En esa laptop, con esos datos, con esa carga. Lo que se traslada es la **comparación** entre dos mediciones hechas en las mismas condiciones.

SECCIÓN 06

Disponibilidad

06

Disponibilidad y los nueves

Disponibilidad *(availability)*: la fracción del tiempo (o de los pedidos) en que el sistema responde bien.

```
make nines

nueves  disponibilidad  caído al año   caído al mes
1       90 %            36.50 días     4320.0 min
2       99 %            3.65 días      432.0 min
3       99.9 %          8.76 h         43.2 min
4       99.99 %         52.56 min      4.3 min
5       99.999 %        5.26 min       0.4 min
```

- Hay que fijar qué es “responder bien” (un 500 no lo es) y sobre qué ventana.
- Cada nueve cambia lo que hay que hacer: con 53 minutos al año, nadie llega a arreglarlo a mano.

En serie se multiplican

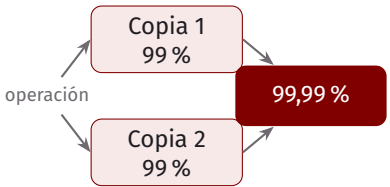



```
node calc.ts series 99.9 99.9 99.9

En serie: 99.7003 %
  caído al año: 26.25 h
  caído al mes: 129.5 min
```

- El conjunto es peor que su peor parte. Cada dependencia en el camino resta.
- Supuesto: las partes fallan de forma independiente.
- La pregunta de diseño: ¿qué necesita *de verdad* la operación para completarse?

En paralelo: la redundancia



$$1 - (1 - 0{,}99)^2 = 0{,}9999$$

- Basta con que responda una. Falla solo si fallan todas a la vez.
- De 3,65 días al año a 53 minutos.
- Aquí la independencia es todo el argumento: dos copias en la misma máquina, con la misma versión, tienden a fallar juntas.
- Y lo que reparte los pedidos entre las copias está en serie con todo.

Trade-off: un nueve más cuesta máquinas, automatización y complejidad. Cuántos nueves es una decisión de negocio.

MTBF y MTTR

Dos tiempos promedio

MTBF: cuánto funciona entre una falla y la siguiente.

MTTR: cuánto tarda en volver.

$$\text{disponibilidad} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

	MTBF	MTTR	
Partida	30 días	43,2 min	99,90 %
Fallar la mitad	60 días	43,2 min	99,95 %
Volver en la mitad	30 días	21,6 min	99,95 %

- Mismo número, distinto costo: prevenir causas raras, o detectar y reiniciar solo.
- Una caída de 8 horas al año y 500 de un minuto dan casi los mismos tres nueves. No son el mismo sistema.

SLI, SLO y SLA: primera mención

- **SLI**: el indicador que se mide. *Fracción de las compras que respondió bien en menos de 1 s.*
- **SLO**: el objetivo que el equipo se fija. *99,5 %, medido cada 30 días.*
- **SLA**: el contrato, con consecuencias si no se cumple.

Un SLO es un escenario de calidad que se vigila en producción. Vuelve en la semana 14.

En la industria

Chubby, el servicio de locks de Google, fallaba tan poco que otros equipos dependían de él como si no pudiera caerse. Si en un trimestre no había tenido una falla real, se provocaba una caída controlada.

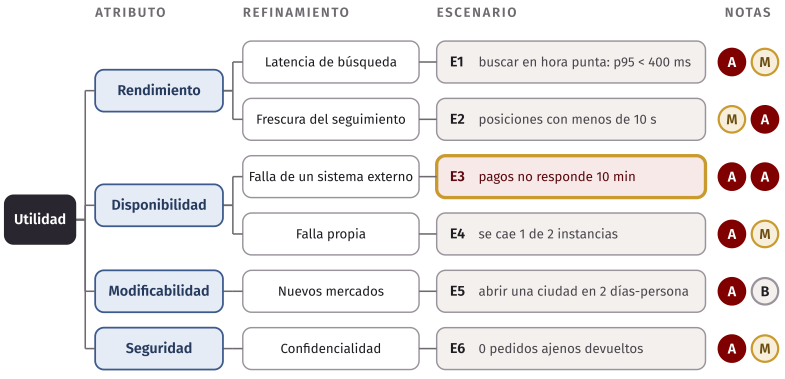
Site Reliability Engineering, cap. 4: sre.google/sre-book/service-level-objectives

SECCIÓN 07

Priorizar y decidir

07

El utility tree



De izquierda a derecha, de lo vago a lo comprobable. Dos notas por hoja: valor para el negocio y dificultad técnica (A, M o B).

E3 es el único (A, A): vale mucho y es difícil. Es el que define la arquitectura.

Ejemplo: Qhatu, pedidos de abarrotes a domicilio.

Qhatu: seis escenarios con sus notas

Atributo	Escenario (resumen)	Negocio	Técnica
Rendimiento	E1. 2 000 compradores buscan en hora punta: p95 < 400 ms, errores < 0,5 %	A	M
Rendimiento	E2. 3 000 repartidores reportan cada 5 s: el 99 % de las posiciones, con menos de 10 s de antigüedad	M	A
Disponibilidad	E3. El procesador de pagos no responde 10 min: toda respuesta en 5 s o menos, 0 pedidos perdidos	A	A
Disponibilidad	E4. Se cae 1 de 2 instancias: errores < 0,5 % en 60 s o menos, 0 pedidos perdidos	A	M
Modificabilidad	E5. Abrir una ciudad: 2 días-persona o menos, 1 módulo, sin interrumpir el servicio	A	B
Seguridad	E6. Un comprador pide pedidos ajenos 1 000 veces: 0 devueltos, alerta en 5 min o menos	A	M

Cada hoja es un escenario completo, con sus seis partes: `scenarios/qhatu.md`.

Cómo se lee

- **(A, A)**: valen mucho y son difíciles. Definen la arquitectura. E3 obliga a decidir cómo se habla con el procesador de pagos.
- **(A, M)**: se diseñan y se comprueban; no deberían dar sorpresas.
- **(M, A)**: antes de pagarlos, se negocia la medida. ¿Aceptarían 30 s en vez de 10?
- **(A, B)**: se hacen, y no condicionan el resto.

Las notas cambian

Son de un momento, de un equipo y de un diseño. El árbol se revisa.

Si todo sale (A, A)

El árbol no priorizó nada. ¿Cuáles dos elegirían si solo pudieran cumplir dos?

El utility tree de Tinkuy es de ustedes: laboratorio, paso 8.

Trade-offs: mejorar un atributo suele costar otro

Decisión	Mejora	Cuesta
Guardar una consulta en una cache	Latencia de lectura	Consistencia: un rato responde con un dato que ya cambió
Agregar una réplica	Disponibilidad y capacidad de lectura	Dinero y operación; la réplica puede ir atrasada
Autenticar y autorizar cada pedido	Seguridad	Latencia y fricción

Un precio de hace 30 segundos casi nunca importa; un asiento que figura libre y ya se vendió, sí. La misma táctica es buena en un escenario y mala en otro.

Decidir es elegir qué se sacrifica, **para este sistema**, y escribirlo.

Los trade-offs que Tinkuy ya tomó

Decisión de v0	Gana	Cuesta
Una conexión nueva a PostgreSQL por consulta	Simplicidad: nada que configurar	Latencia, y un límite: 100 conexiones
El cobro y el correo dentro de la transacción	Todo se confirma o se deshace junto	Latencia y disponibilidad: la compra necesita a los dos
Un proceso y una base de datos	Se levanta con dos comandos	Todo falla junto y escala junto

Ninguna es un error en sí misma: con diez ventas al día, las tres se defienden.
Que sean un problema depende de los escenarios que lleguen.

Cómo se escribe un trade-off

La forma

Para cumplir **[escenario]**, elegimos **[decisión]** en vez de **[alternativa]**. Ganamos **[medida que mejora]**. Aceptamos **[medida que empeora, y en qué escenario]**.

Qhatu, E3

Para cumplir E3, elegimos esperar al procesador de pagos 3 segundos como máximo y ofrecer pago contra entrega, en vez de esperar hasta que responda. Ganamos que todo intento de pago recibe respuesta en 5 segundos o menos. Aceptamos que, en un día normal, algunos pagos que habrían terminado bien en 4 segundos se den por fallidos.

Escenario, decisión, alternativa y costo. En la semana 7, esto es un ADR.

Laboratorio: escenarios para Tinkuy, medidos con k6

1. Los tres candidatos a ASR, como escenarios con el template. Intercambiar con otra pareja.
2. `make load-browse`: leer el resumen. Anotar los números y la máquina.
3. Tres corridas: ¿cuánto varían el p95 y el promedio? Una regla para comparar.
4. `make endpoints`: el endpoint lento, y por qué, leyendo su código.
5. Un umbral en una copia de `load/browse.js`: la medida como prueba.
6. `make load-sale` y `make oversell`: ¿qué atributo se incumple? Escribir el escenario.
7. Disponibilidad en serie: Tinkuy, la pasarela y el correo.
8. El utility tree de Tinkuy: seis escenarios, tres atributos, los (A, A).

P0

Los escenarios y las mediciones de hoy son la segunda parte de P0.

Resumen

- La funcionalidad no determina la arquitectura; los atributos de calidad sí.
- Un atributo se escribe como **escenario de seis partes**. Sin medida no es un escenario; sin entorno, la medida no dice nada.
- La latencia se resume con **percentiles**, no con el promedio. No se promedian.
- Medir sin engañarse: repetir, separar, medir desde quien llama y cuidar la **omisión coordinada**.
- El umbral sale del stakeholder; de la laptop se traslada la **comparación**.
- **Disponibilidad**: los nueves; en serie se multiplican; fallar menos o recuperarse antes.
- El **utility tree** deja arriba los (A, A). Un **trade-off** dice qué se sacrifica, y se escribe.

Para la próxima clase

- Leer: David Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules* (1972). Son seis páginas: completo.
- Leer: *Software Architecture in Practice*, cap. 8 *Modifiability*.
- Terminar el laboratorio: escenarios, mediciones y utility tree de Tinkuy entran en P0.

Lecturas de esta semana

Software Architecture in Practice, cap. 3 y cap. 19 (utility tree); Barbacci y otros, *Quality Attributes* (SEI, 1995); *Designing Data-Intensive Applications* (2.ª ed.), cap. 2; *Site Reliability Engineering*, cap. 4 *Service Level Objectives*. Complementarias: ISO/IEC 25010; k6, *Metrics* y *Thresholds*.

Pregunta: agregar un medio de pago a Tinkuy, ¿cuántos archivos toca hoy? ¿De qué depende ese número?