Adventures in zero-downtime

… or, I thought this would just work?

https://github.com/bittrance/zero-downtime-demo

The corrosive effect of deploy errors

Our clients experience errors when we deploy, so we …

… should deploy less often

… must communicate in advance when we deploy

… will monitor deploys

… see less meaning in automation

Zero-downtime is necessary

for successful continuous delivery

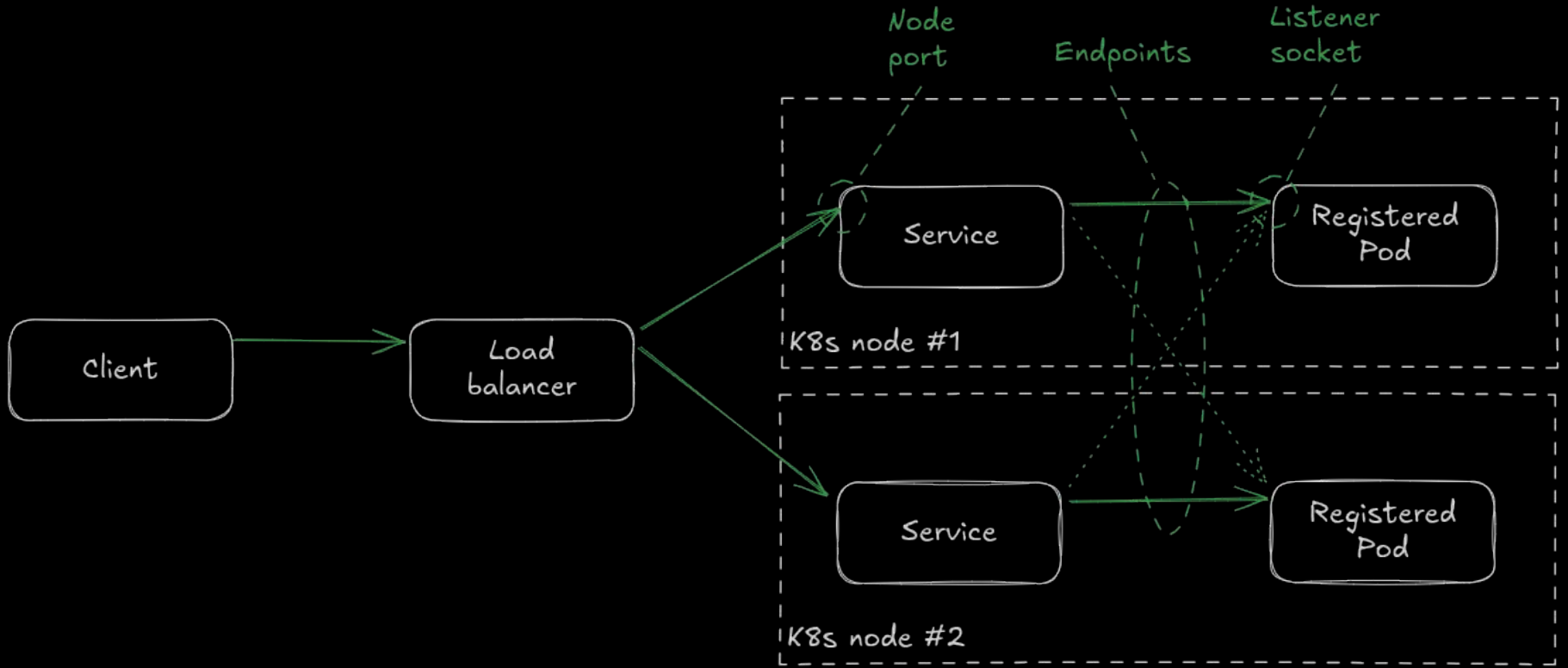
First, terminology

zero downtime - service maintenance is not directly observable by users

graceful shutdown - terminating a process without disrupting in-flight requests

persistent connection - a long-running (typically TCP) connection reused between requests

A typical Kubernetes deployment



A resilient service on Kubernetes

Must have:

- replicas > 1
- RollingUpdate strategy
- container probes

Nice to have:

- topology spread and anti-affinity
- pod disruption budget

```yaml
spec:
  replicas: 5
  selector:
    matchLabels:
      app.kubernetes.io/name: hello-rest
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app.kubernetes.io/name: hello-rest
    spec:
      containers:
        - name: api
          image: bittrance/hello-world:spring-undertow-hello-rest
          startupProbe:
            failureThreshold: 10
            httpGet:
              path: /health
              port: 8080
            periodSeconds: 1
            successThreshold: 1
            timeoutSeconds: 1
          readinessProbe:
```

Graceful shutdown

Pods that shut down slowly should not continue to serve regular traffic and should start terminating and finish processing open connections. Some applications need to go beyond finishing open connections and need more graceful termination, for example, session draining and completion.

https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

```
FROM python:3.13-slim

WORKDIR /app
COPY app.py requirements.txt .
RUN pip install -r requirements.txt
USER nobody
ENTRYPOINT ["gunicorn"]
EXPOSE 3000
CMD ["--workers=4", "--graceful-timeout=60", "app:build()"]
```

```
let listener = tokio::net::TcpListener::bind("0.0.0.0:8080").
↪ await.unwrap();
axum::serve(listener, app)
    .with_graceful_shutdown(shutdown_signal())
    .await
    .unwrap();
```

So how do we test it?

Hello world spring-boot app

- single controller
- openjdk 21
- webserver undertow

```java
@RestController
public class HomeResource {
    @Value("${HELLO_REST_REQUEST_DELAY:1.0}")
    private float requestDelay;

    @GetMapping("/health")
    public String health() {
        return "ok";
    }

    @GetMapping("/")
    public String index() throws InterruptedException {
        long delay = Float.valueOf(this.requestDelay * 1000).
        ↪ longValue();
        Thread.sleep(delay);
        return "Hello World!\n";
    }
}
```

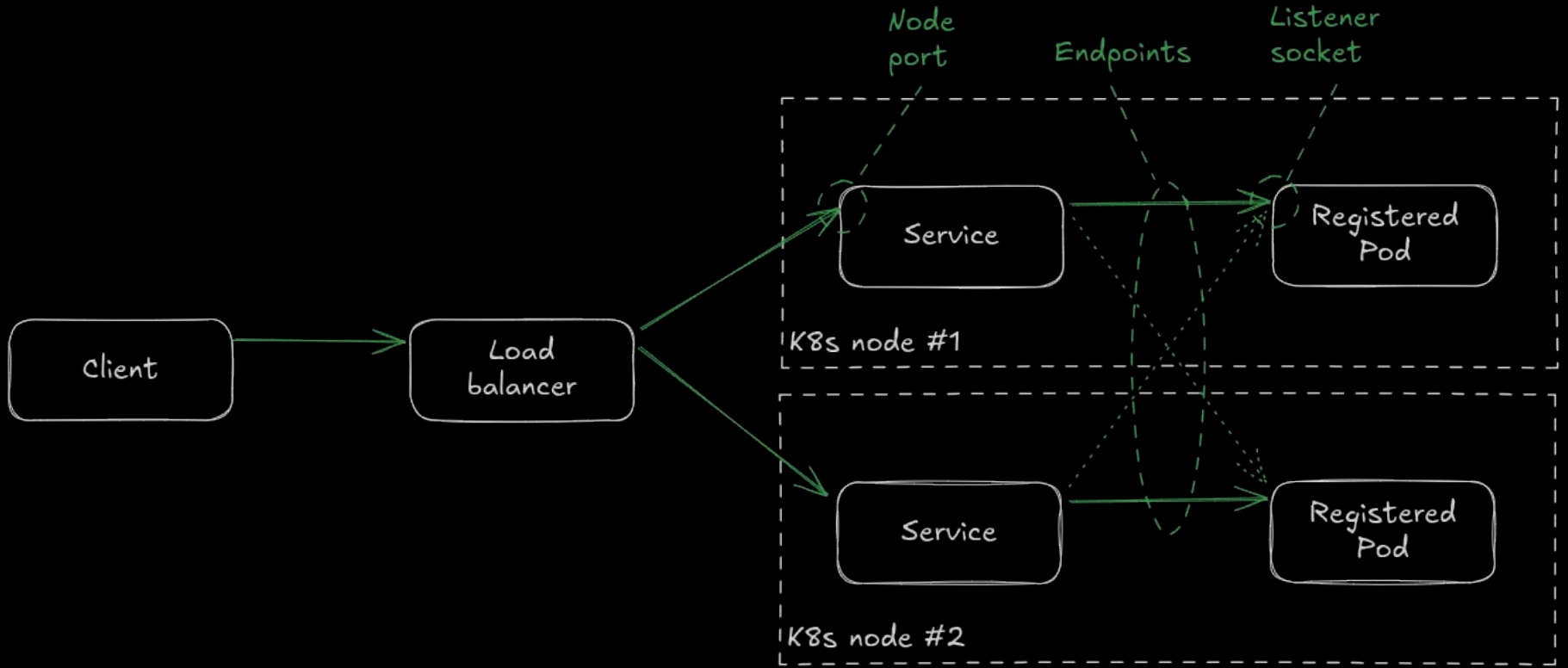
A simple k6 load test

- github.com/grafana/k6
- synchronous js engine
- extensive ecosystem

```
import http from "k6/http";
import { check } from "k6";

const ENDPOINT = __ENV.HELLO_REST_ENDPOINT || "http://localhost:8080";

export const options = {
  scenarios: {
    get_greeting: {
      executor: "constant-arrival-rate",
      rate: 200,
      duration: "30s",
      timeUnit: "1s",
      preAllocatedVUs: 100,
    }
  },
  summaryTrendStats: ['min', 'med','p(80)', 'p(99)', 'max', 'count'],
}

export default function() {
  let res = http.get(ENDPOINT);
  if(res.status != 200) {
    console.log(`Endpoint returned ${res.status}: ${res.body}`);
  }
  check(res, {
    "status is 200": (r) => r.status == 200,
  });
}
```

Shell time!

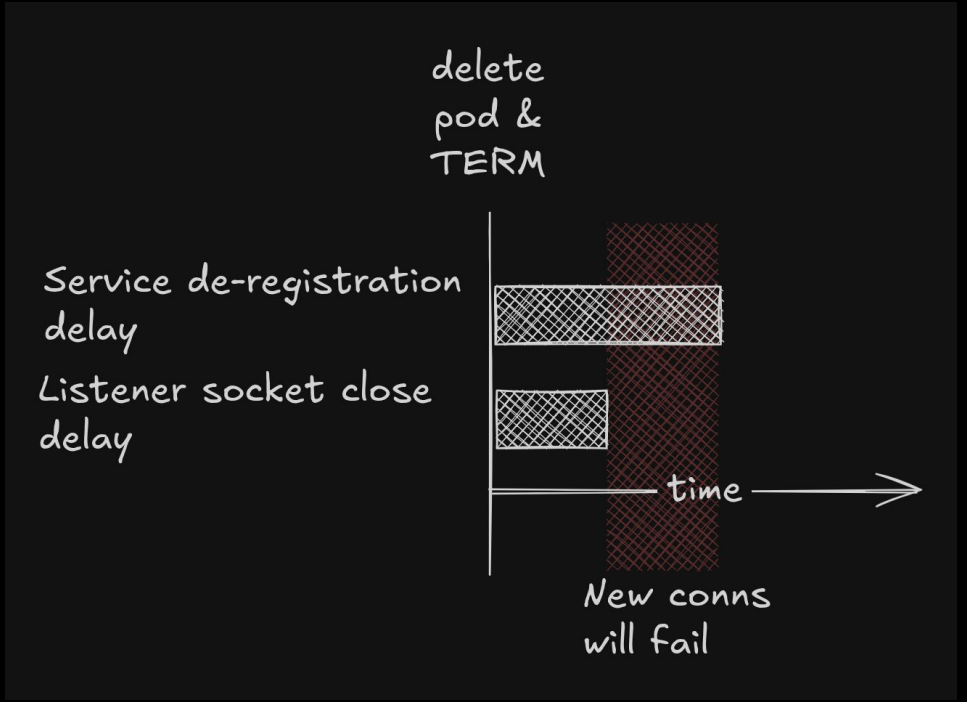
A typical Kubernetes deployment



Problem: service de-registration delay

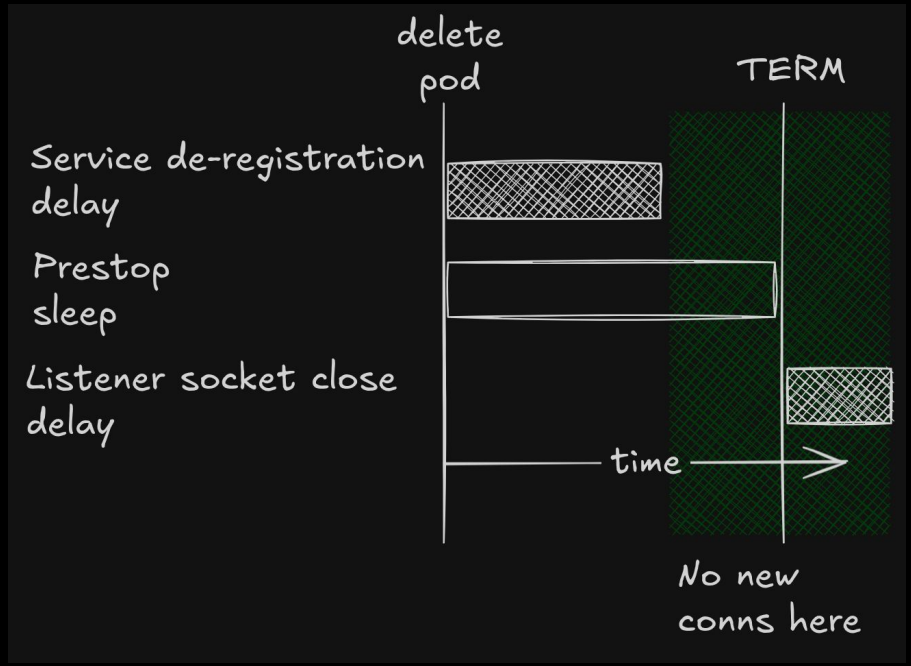
*Any endpoints that represent the terminating Pods are **not immediately** removed from EndpointSlices, and a status indicating terminating state is exposed from the EndpointSlice API.*

https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/



Solution: pre-stop sleep

```yaml
template:
  metadata:
    labels:
      app.kubernetes.io/name: hello-rest
  spec:
    containers:
      - name: api
        image: bittrance/hello-world:spring-undertow-hello-rest
        lifecycle:
          preStop:
            sleep:
              seconds: 2
        startupProbe:
          failureThreshold: 10
          httpGet:
            path: /health
            port: 8080
          periodSeconds: 1
          successThreshold: 1
          timeoutSeconds: 1
        readinessProbe:
```



Does it work this time?

But, persistent connections?

HTTP/1.1 with header Connection: keep-alive

HTTP/2 is fully multiplexed, each request a “stream”

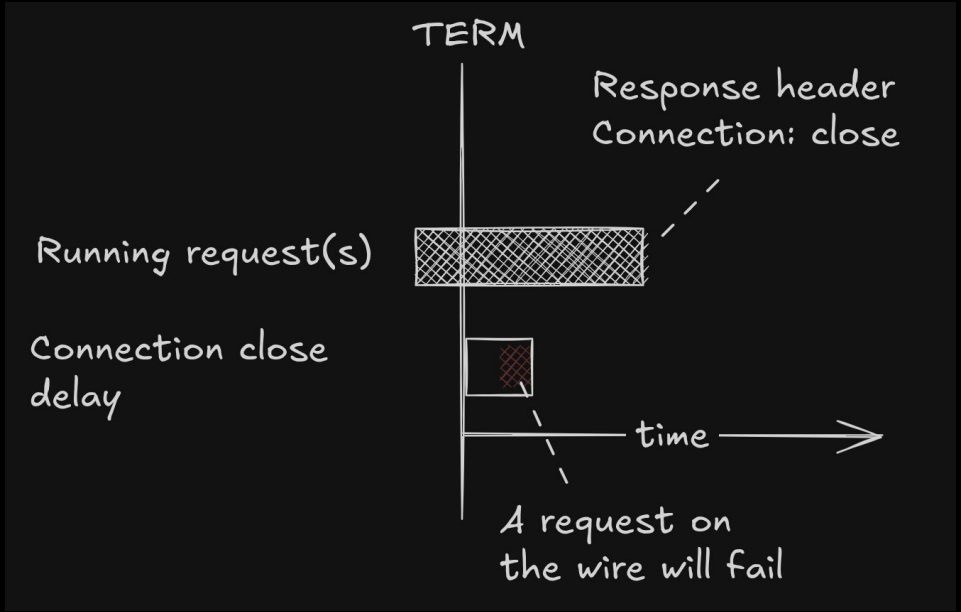
GRPC (v3) is HTTP/2

Strategy one: immediate hang-up

Plus: fast, likely before p(95) latency

Minus: small risk of losing requests

Use when you have many persistent connections with few requests each.

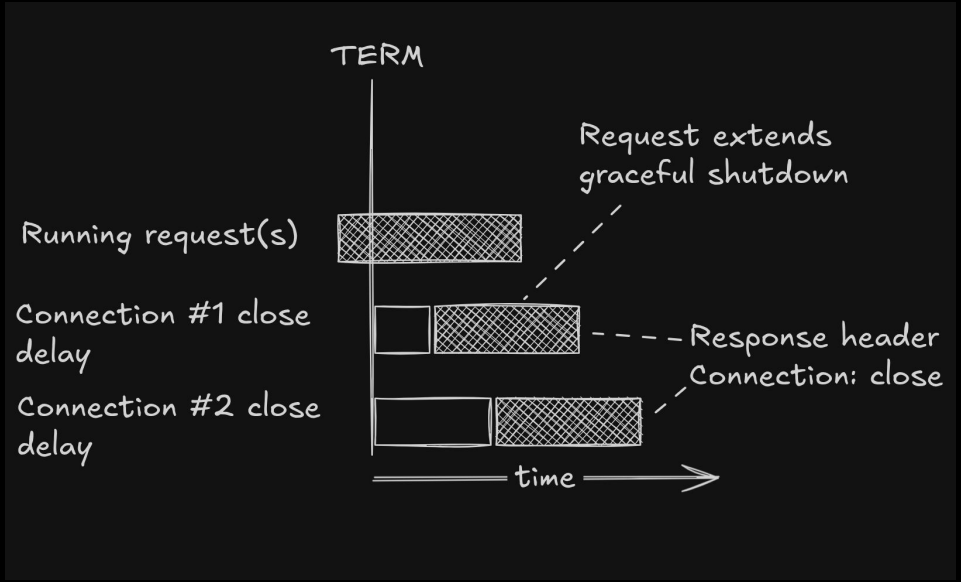


Strategy two: one more message

Plus: no requests are lost (in HTTP/1.1)

Minus: may be protracted

Use when requests arrive via round-robin L7 load balancing.



Once more, with persistence!

But we’re using node.js!

Node express app

server.close([callback]) stops the server from accepting new connections and closes all [idle] connections.

https://nodejs.org/api/http.html

… but only v19+

```
import http from 'node:http';
import express from 'express';

const delay = parseFloat(
  process.env['HELLO_REST_REQUEST_DELAY'] || '1.0'
) * 1000;

const app = express();
app.get('/', (req, res) => {
  setTimeout(() => res.end('Hello World!'), delay);
});

const opts = {keepAliveTimeout: 10000};
const server = http.createServer(opts, app);
server.listen(8080);

process.on('SIGINT', () => {
  server.close(() => console.log("closed!"));
});
```

So we’re cool, right?

Node express + terminus

- github.com/godaddy/terminus
- correct immediate shutdown
- signal handling
- also health check routes

```
import http from 'node:http';
import express from 'express';
import { createTerminus } from '@godaddy/terminus';

const delay = parseFloat(
  process.env['HELLO_REST_REQUEST_DELAY'] || '1.0'
) * 1000;

const app = express()
app.get('/', function (req, res) {
  setTimeout(() => res.send('Hello World!'), delay);
})

const opts = {keepAliveTimeout: 10000};
const server = http.createServer(opts, app);
server.listen(8080);

createTerminus(server, {
  signals: ['SIGTERM', 'SIGINT'],
  useExit0: true,
  timeout: 10000,
});
```

But, we’re using Go!

Go Hello World service

Shutdown gracefully shuts down the server without interrupting any active connections. Shutdown works by first closing all open listeners, then closing all idle connections, and then waiting indefinitely for connections to return to idle and then shut down.

- http.Server Shutdown docs

```go
func main() {
    r := gin.Default()
    r.GET("/", func(c *gin.Context) {
        c.String(http.StatusOK, "Hello world!")
    })

    server := &http.Server{ Addr: ":8080", Handler: r }

    shutdownComplete := make(chan struct{})
    go func() {
        quit := make(chan os.Signal, 1)
        signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
        <-quit

        grace_period := time.Duration(5.0 * float64(time.Second))
        ctx, cancel := context.WithTimeout(context.Background(),
        ↪grace_period)
        defer cancel()
        if err := server.Shutdown(ctx); err != nil {
            log.Printf("HTTP server Shutdown: %v", err)
        }
        close(shutdownComplete)
    }()

    if err := server.ListenAndServe(); err != nil && err != http.
    ↪ErrServerClosed {
        log.Fatalf("listen: %s\n", err)
    }
    <-shutdownComplete
}
```

Yes, it works!

All tested frameworks

Lessons

- HTTP/1.1 is from 1999 - we’re still not getting it right
- understand your networking context
- latency and rate matters
- long-polling is the exception

Bonus lesson:

- proxy sidecar + persistent connections is hard to get right

Questions?