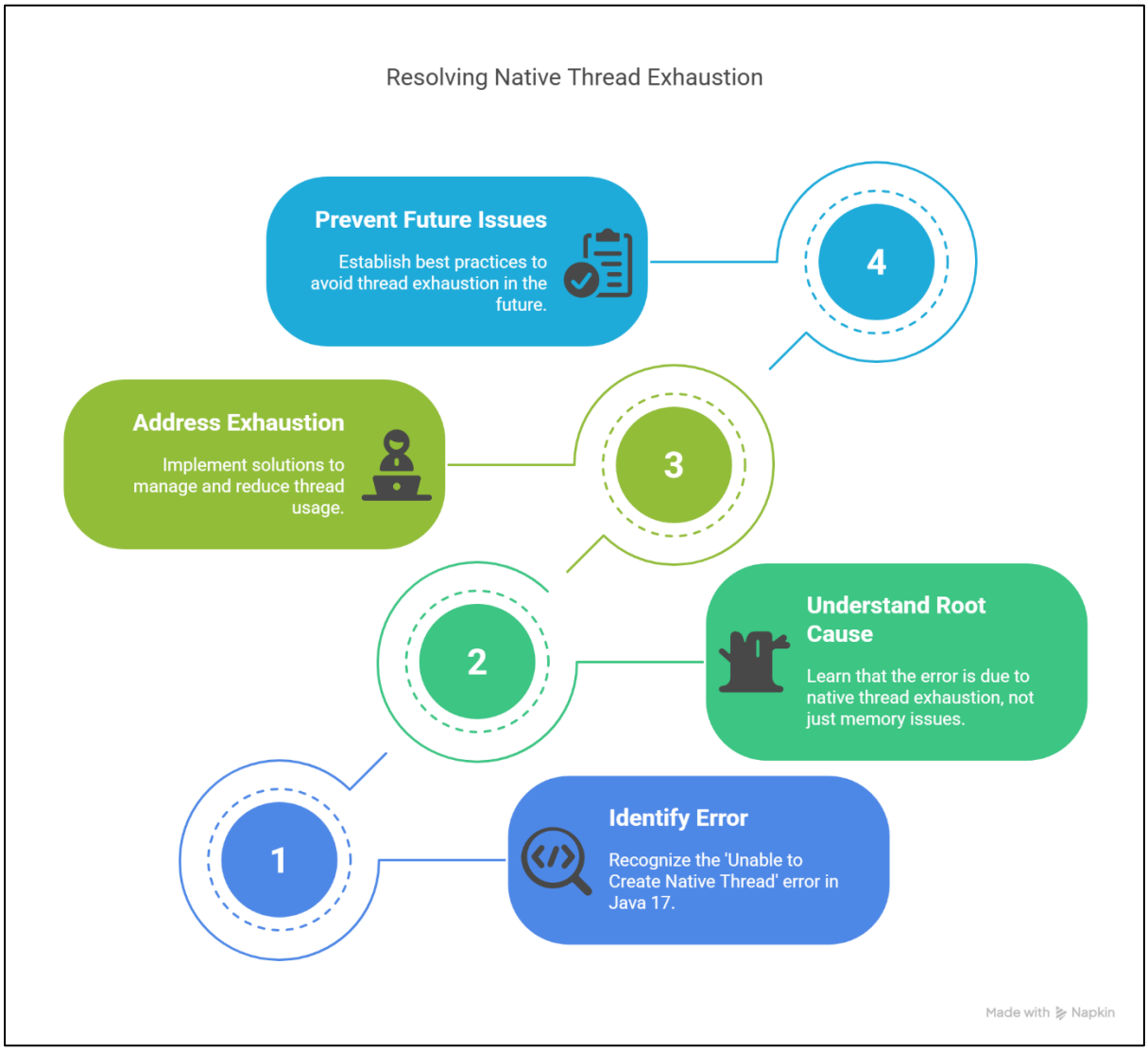
🚨 Resolving the ‘java.lang.OutOfMemoryError: Unable to Create Native Thread' After Migrating to Java 17 🛠️

🧠 Problem Statement:

After migrating to Java 17, teams often encounter the dreaded java.lang.OutOfMemoryError: unable to create native thread. This error halts applications, especially in containerized environments or high-throughput systems, despite no prior issues in older Java versions. The root cause? **Java 17’s stricter memory management, OS resource limits, and thread-handling changes**. Let’s decode why this happens and how to fix it!

After migrating to **Java 17**, you’re not alone.

This error is **not just about memory** — it’s a **symptom of native thread exhaustion**, and unless you address the root cause, your application might fail under load.



📌 What's Going Wrong?

Each `java.lang.Thread` in Java maps to an **OS-level native thread**, which consumes:

- OS-level stack memory (1MB–2MB per thread)
- Limited system resources (max processes/threads per user)
- CPU context-switching power

And when you hit system/thread limits (ulimit, PID, memory), the JVM says:

❌ *"Unable to create native thread"*

🔍 Root Cause in Most Modern Apps?

- ❗ Repeated creation of `CloseableHttpClient` or thread pools
- ❗ Schedulers or APIs spawning new threads every run
- ❗ Libraries with internal thread spawners (like `IdleConnectionEvictor`)
- ❗ Not reusing or shutting down threads
- ❗ No pooling, no eviction policy

These are **silent killers**—you may not notice until production crashes.

🛠️ How to Detect It – Like a Pro

✅ Use OS-Level Commands

```
ps -eLf | grep java
top -H -p <PID>
ulimit -u  # max user processes
```

✅ Use JVM Tools

- `jstack <PID>` → inspect growing thread patterns
- `jcmd <PID> Thread.print`
- VisualVM / JMC / YourKit → see real-time thread creation & GC stats

Look for repeating thread names:

- `HttpClient-Connection-*`
- `IdleConnectionEvictor-*`
- `scheduler-*`
- `pool-*`

💣 Common Pitfall (Real-World Code)

```
@Scheduled(fixedRate = 60000)
public void pingAPI() {
    try (CloseableHttpClient client = HttpClients.createDefault()) {
        // BOOM: New client, new threads, every minute
    }
}
```

Each execution creates a new thread pool under the hood.

✅ Fix it Like a Java Architect

Spring Bean Configuration (Singleton) ✅

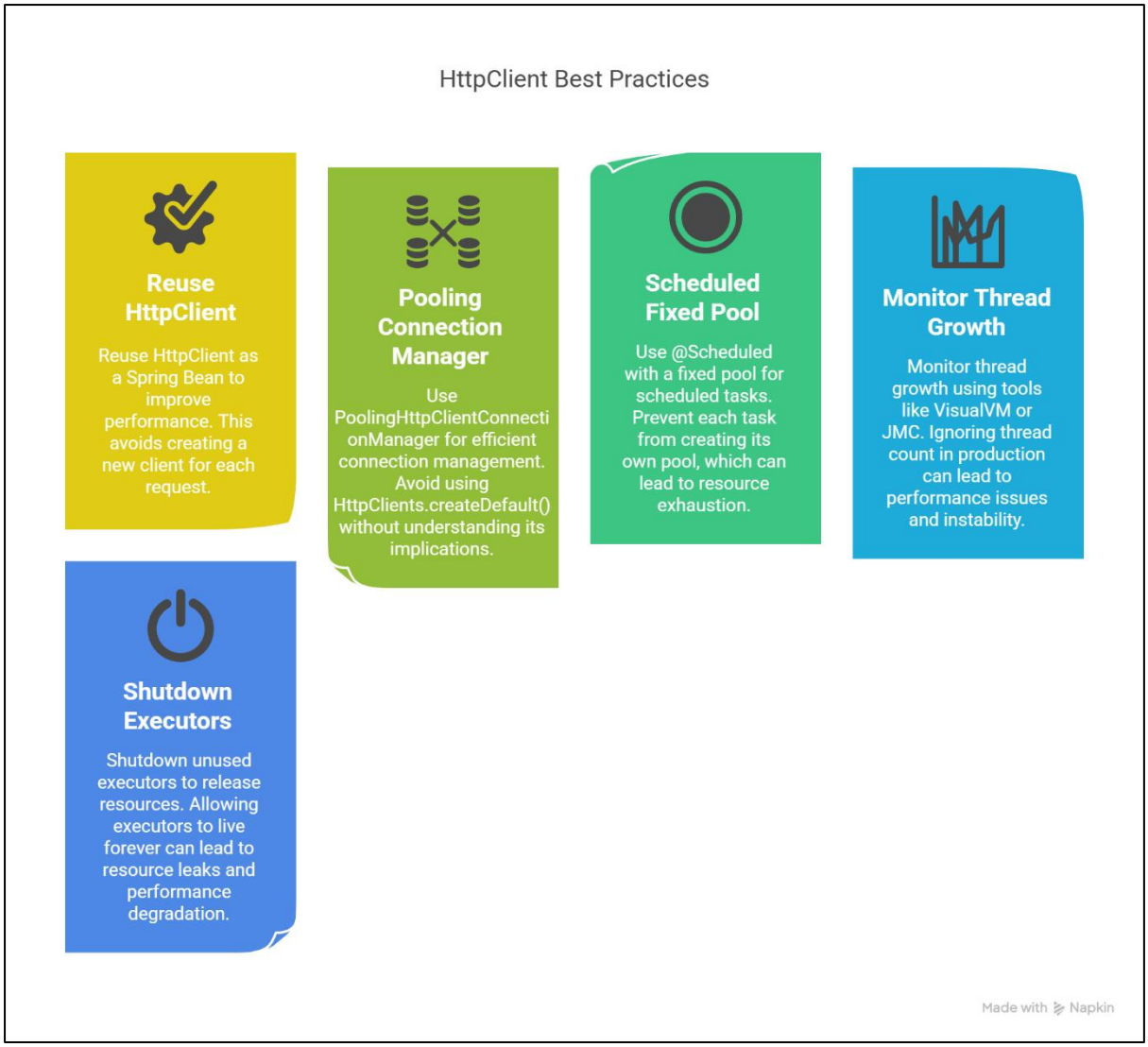
```
@Bean
public CloseableHttpClient httpClient() {
    PoolingHttpClientConnectionManager connManager =
        PoolingHttpClientConnectionManagerBuilder.create()
            .setMaxConnTotal(100)
            .setMaxConnPerRoute(20)
            .build();

    return HttpClients.custom()
        .setConnectionManager(connManager)
        .evictIdleConnections(TimeValue.ofMinutes(5))
        .build();
}
```

And then **inject it wherever needed—don't recreate it per call**.

📊 Best Practices Checklist ✅

✅ Do This	❌ Avoid This
Reuse `HttpClient` as a Spring Bean	Create new `HttpClient` per request
Use `PoolingHttpClientConnectionManager`	Use `HttpClients.createDefault()` blindly
Use `@Scheduled` with a fixed pool	Let each task create its own pool
Monitor thread growth (VisualVM, JMC)	Ignore thread count in production
Shutdown unused executors	Let executors live forever



🧪 Bonus: Log Thread Creation for Debugging

```
 ThreadFactory loggingFactory = r -> {
            Thread t = new Thread(r);
            t.setName("CustomPoolThread-" + t.getId());

            // Log thread creation with a stack trace
            System.out.println("📝 Thread created: " + t.getName());
            new Exception("📌 Thread creation
trace:").printStackTrace(System.out);

            return t;
        };

        // Create an ExecutorService using the logging ThreadFactory
        ExecutorService executor = Executors.newFixedThreadPool(2,
loggingFactory);
```

Use this with your executor services to **trace who is creating threads and when.**

🧠 Final Thoughts

This error **isn't always about heap space**. It's a **thread management problem**—which got **amplified with Java 17 changes**, container limits, and more concurrency-oriented APIs.

By managing thread pools wisely, analysing thread dumps, and **using Spring beans for connection-heavy components**, you can avoid thread leaks and **make your app production hardened**.

👇 Drop your experience or questions in the comments.

Syed Shahul Hameed