ChatGPT

Java Microservices Anti-Patterns Guide

1. Hardcoded Configuration Values

Hardcoding configuration such as URLs, file paths, or magic numbers in code is a common anti-pattern. It ties the code to specific environments and makes it difficult to adjust behavior without modifying and redeploying the service. In a microservices context, this is especially problematic because services often need to be configured differently for development, testing, and production (e.g., different database URLs or service endpoints). Hardcoded values reduce flexibility and can lead to errors when an environment changes.

Anti-Pattern Example – Hardcoded Config: In the code below, the database URL and timeout are fixed in the source. This makes changing the DB host or tuning timeout impossible without code changes. It also exposes environment details in code:

```
public class UserRepository {
    // Anti-pattern: hardcoded configuration values
    private String DB_URL = "jdbc:mysql://localhost:3306/prod_db";
    private int TIMEOUT_SECONDS = 30;

    public Connection connect() throws SQLException {
        // Using hardcoded DB URL and timeout
        DriverManager.setLoginTimeout(TIMEOUT_SECONDS);
        return DriverManager.getConnection(DB_URL, "dbuser", "secret");
    }
}
```

Improved Solution – Externalized Config: Here, configuration is loaded from external sources (environment variables or config files). This allows different values per environment and avoids recompiling code for configuration changes. The code uses dependency injection or a config utility to fetch values, making the service more flexible and portable:

```
public class UserRepository {
    // Good practice: load configuration from environment or config service
    private String dbUrl;
    private int timeoutSeconds;

    public UserRepository(Config config) {
        this.dbUrl = config.get("DB_URL");              // e.g., "jdbc:mysql://
dbserver:3306/prod_db"
        this.timeoutSeconds = config.getInt("TIMEOUT"); // e.g., 30
```

```
    }

    public Connection connect() throws SQLException {
        DriverManager.setLoginTimeout(timeoutSeconds);
        return DriverManager.getConnection(dbUrl, config.get("DB_USER"),
config.get("DB_PASSWORD"));
    }
}
```

Why It's Better: Externalizing configuration (via environment variables, property files, or a configuration server) enables changing settings without altering code. In microservices, this means you can deploy the same artifact to different environments with different settings. It improves maintainability and follows the *12-Factor App* principle of strict separation of config from code. The improved example shows using a `Config` helper (could be a framework's `@Value` or similar in Spring) to retrieve settings. This makes the service adaptable and easier to manage across environments.

2. Hardcoded Credentials and Secrets

Embedding sensitive credentials (passwords, API keys, tokens) directly in code is a serious anti-pattern. It risks security (exposing secrets in source control) and makes rotation or configuration of secrets cumbersome. In microservices which might be open-sourced or containerized, having secrets in code can lead to accidental leaks and difficulties in managing different credentials for dev/test/prod.

Anti-Pattern Example – Hardcoded Secret: The API key is directly in the code. This might end up in version control and cannot be changed without code changes, posing security and flexibility issues:

```
public class PaymentService {
    // Anti-pattern: hardcoded API secret in code
    private static final String PAYMENT_API_KEY = "ABC123SECRETKEY";

    public PaymentClient createClient() {
        return new PaymentClient(PAYMENT_API_KEY);
    }
}
```

Improved Solution – Externalized Secret Management: In the corrected example, the secret is pulled from a secure source at runtime (e.g., environment variable, vault, or config server). The code does not contain the actual secret, reducing risk and allowing the secret to be changed or rotated without modifying the code:

```
public class PaymentService {
    // Good practice: retrieve secret from environment or secure vault
    private final String paymentApiKey;
```

```
    public PaymentService() {
        this.paymentApiKey = System.getenv("PAYMENT_API_KEY");
        // Alternatively, use a secrets manager or config service
    }

    public PaymentClient createClient() {
        return new PaymentClient(paymentApiKey);
    }
}
```

Why It's Better: By not hardcoding credentials, we improve security and flexibility. Secrets can be managed outside the code (for example, injected via environment variables in Kubernetes or retrieved from a secret manager). This way, leaking of secrets in source code is prevented and operations teams can rotate keys or change credentials without touching the source code. This practice is crucial in distributed architectures, as each microservice might have its own credentials for databases or external APIs.

3. Manual Dependency Instantiation (Not Using DI)

Manually creating dependent objects inside a class using `new` or factory lookups is an anti-pattern in modern Java development, especially when frameworks supporting Dependency Injection (DI) are available (e.g., Spring). It leads to tight coupling between classes and makes testing or replacing dependencies difficult. In a microservice, failing to use DI can result in code that is rigid and hard to maintain or configure.

Anti-Pattern Example – Manual Object Creation: In this example, the `OrderService` manually creates an `EmailService`. This hard-coded dependency means `OrderService` always uses that specific implementation, and it's difficult to substitute a different notification mechanism or mock it in tests:

```
public class OrderService {
    // Anti-pattern: manually instantiating dependencies
    private EmailService emailService = new EmailService();

    public void placeOrder(Order order) {
        // ... order processing logic ...
        emailService.sendConfirmation(order.getCustomerEmail());
    }
}
```

Improved Solution – Dependency Injection: Here, `OrderService` receives its dependencies via constructor (constructor injection). It depends on an interface `NotificationService` rather than a concrete `EmailService`. This approach (commonly used with frameworks or manually) makes it easy to substitute implementations (e.g., an `SmsService`) and to unit test by injecting mocks:

```
public interface NotificationService {
    void notifyCustomer(String contact);
```

```
}

public class EmailService implements NotificationService {
    public void notifyCustomer(String email) {
        // send email confirmation
    }
}

// Using constructor injection for dependency
public class OrderService {
    private final NotificationService notificationService;

    public OrderService(NotificationService notificationService) {
        this.notificationService = notificationService;
    }

    public void placeOrder(Order order) {
        // ... order processing logic ...
        notificationService.notifyCustomer(order.getCustomerEmail());
    }
}

// Elsewhere, configuration wires the dependency, e.g.:
OrderService svc = new OrderService(new EmailService());
```

Why It's Better: Using dependency injection (explicitly or via a framework container) loosens coupling. `OrderService` depends on an abstraction (`NotificationService` interface) and doesn't control the concrete implementation. This improves testability (we can pass a mock or stub for `NotificationService`), and flexibility (swap out email for SMS or other mechanisms without changing `OrderService`). In microservices, DI also centralizes object creation, which is helpful for managing configurations like pooling, transactions, or proxies around dependencies.

4. Service Locator (Hidden Dependency Lookup)

The Service Locator pattern, where code looks up dependencies from a registry or context, is considered an anti-pattern in modern DI-based systems. It hides a class's dependencies and makes the code depend on the locator, leading to less transparent and harder-to-test code. In a microservice using frameworks like Spring, directly querying the application context for beans is usually a bad practice.

Anti-Pattern Example – Service Locator: The code fetches a bean from a global application context. This makes `AuthController` implicitly dependent on the context and obscures what it needs. It also complicates testing because the context must be available or mocked:

```
public class AuthController {
    // Anti-pattern: using a service locator to get a bean
    public String login(String user, String pass) {
```

```
        UserService userService = AppContext.getBean(UserService.class);
        // AppContext is a hypothetical global context or similar to Spring's
ApplicationContext
        if (userService.validateCredentials(user, pass)) {
            return userService.generateToken(user);
        }
        throw new RuntimeException("Invalid login");
    }
}
```

Improved Solution – Dependency Injection (Direct Injection): In the improved version, `AuthController` clearly declares its dependency on `UserService` via constructor (or field) injection. The framework (or manual wiring) supplies the `UserService`, eliminating the need for a lookup. The code is clearer about its collaborators:

```
public class AuthController {
    private final UserService userService;

    // Dependency is injected via constructor
    public AuthController(UserService userService) {
        this.userService = userService;
    }

    public String login(String user, String pass) {
        if (userService.validateCredentials(user, pass)) {
            return userService.generateToken(user);
        }
        throw new RuntimeException("Invalid login");
    }
}
```

Why It's Better: The improved approach makes dependencies explicit. `AuthController` is no longer responsible for locating its dependency; instead, it's given to it. This results in code that is easier to understand (less “magic” happening behind the scenes), and easier to test (we can directly instantiate `AuthController` with a mock `UserService`). Avoiding service locator ensures that microservices remain modular and that configuration of dependencies is handled in one place (the composition root or framework), rather than scattered throughout the codebase.

5. Swallowing Exceptions (Ignoring Errors)

Failing to properly handle exceptions — for instance, catching an exception and doing nothing or simply logging without action — is an anti-pattern. Swallowed exceptions can make debugging and error recovery very difficult, especially in microservices where a failure in one service might need to be communicated back to the client or trigger compensating actions. Ignoring exceptions can lead to inconsistent state or silent failures.

Anti-Pattern Example – Silently Ignoring Exceptions: The code catches a generic exception and does nothing (or only prints the stack trace to console). This means the error is effectively lost or not handled appropriately. The user or calling service will not know that something went wrong, and the system might continue in a bad state:

```
public void processMessage(String msg) {
    try {
        // parse and process the message
        handleMessage(msg);
    } catch (Exception e) {
        // Anti-pattern: swallowing the exception
        System.err.println("Error processing message: " + e.getMessage());
        // e.printStackTrace();  (even printing to console is not sufficient
handling)
        // (No rethrow or proper handling)
    }
    // execution continues as if nothing happened
}
```

Improved Solution – Handle or Propagate Exceptions: In the fixed example, we catch only expected exceptions and handle them or wrap them in a custom exception after logging. Unexpected exceptions are propagated (not caught at this level) so they can be handled by a global error handler or result in proper error responses. This ensures the failure is visible and can be dealt with:

```
public void processMessage(String msg) {
    try {
        handleMessage(msg);
    } catch (InvalidMessageFormatException e) {
        // Handle a specific, expected exception
        log.error("Invalid message format: " + e.getMessage());
        // Perhaps send to a dead-letter queue or respond with a specific error
        throw new ProcessingException("Failed to process message", e);
    } catch (Exception e) {
        // Catch-all for any other unforeseen exceptions
        log.error("Unexpected error processing message", e);
        throw e; // rethrow to ensure it isn't suppressed
    }
}
```

Why It's Better: The improved code doesn't simply hide exceptions. By catching specific exceptions, we can take remedial actions or translate them into meaningful errors (e.g., return an HTTP 400 Bad Request if input is invalid). Logging the exception (with stack trace) ensures visibility for debugging. Propagating or rethrowing unexpected exceptions prevents the system from running in a corrupt state and allows higher-level handlers (like an exception mapper or global error handler in a microservice) to generate an appropriate failure response. In microservices, this is vital to maintain reliability and make failures

observable (e.g., via logs or monitoring). Simply ignoring exceptions could lead to cascading failures that are very hard to trace.

6. Overly Broad Exception Catch

Catching exceptions at too high a level (e.g., catching `Exception` or `Throwable`) is an anti-pattern because it can unintentionally catch and suppress important errors that you did not intend to handle. In microservices, overly broad catches might intercept runtime exceptions like OutOfMemoryError or database connection issues and mask them, making the service fail in unpredictable ways. It also makes it harder to know what actually went wrong, since everything funnels through one catch block.

Anti-Pattern Example – Catching All Exceptions: The code below catches `Exception`, meaning it will catch anything thrown inside the try, including runtime exceptions that might be better handled elsewhere or cause a crash. By converting everything to a generic error message, it loses the original exception context:

```
public String getUserProfile(String userId) {
    try {
        // Possibly throwing various exceptions (e.g., NullPointerException,
SQLException)
        return userDao.fetchProfile(userId);
    } catch (Exception e) {
        // Anti-pattern: catching too broadly and generalizing error
        log.error("Error fetching profile", e);
        return "ERROR";  // generic fallback, no clear handling
    }
}
```

Improved Solution – Catch Specific Exceptions: Catch only the exceptions you expect and know how to handle. Let others propagate. In this example, we catch a specific checked exception (like a `SQLException` from the DAO) and handle it, perhaps returning a not-found result or a default. We allow runtime exceptions to bubble up, or catch them separately if we have a specific strategy (like wrapping into a custom unchecked exception):

```
public String getUserProfile(String userId) {
    try {
        return userDao.fetchProfile(userId);
    } catch (SQLException e) {
        // Handle a known exception (e.g., database error or no data)
        log.error("Database error fetching profile for " + userId, e);
        return null; // or throw custom exception indicating failure
    } catch (UserNotFoundException e) {
        // Perhaps a custom exception the DAO might throw
        log.warn("Profile not found for user " + userId);
        return null;
```

```
    }
}
```

Why It's Better: This approach handles only what it can meaningfully recover from. Catching specific exceptions keeps error handling logic focused and avoids accidentally suppressing other issues. If an unexpected exception (like `NullPointerException`) occurs, it will propagate up and likely be caught by a global error handler (resulting in a 5xx response or similar). This is preferable to catching it and returning a generic "ERROR" or null, which could lead to downstream confusion. In microservices, clarity in error handling ensures that each service fails clearly and does not hide problems that could later cause data inconsistencies or mysterious behavior.

7. Using Exceptions for Flow Control

Using exceptions to manage normal program flow (often known as “exception-driven logic”) is an anti-pattern. Exceptions are meant for error conditions, not regular control flow, and using them as such can lead to code that is hard to read and perform poorly (since throwing and catching exceptions is relatively expensive). In a microservices environment, abuse of exceptions can also lead to confusing error logs and difficulty in distinguishing actual errors from expected behavior.

Anti-Pattern Example – Exception as Logic: In this code, an exception is thrown to break out of a search loop when an item is found. This is using an exception to control the loop rather than as an error indicator, which is a misuse of exceptions:

```
public Item findItem(List<Item> items, String id) {
    try {
        items.forEach(item -> {
            if (item.getId().equals(id)) {
                // Anti-pattern: throwing exception to control flow (to escape
from lambda)
                throw new FoundException(item);
            }
        });
    } catch (FoundException found) {
        // Caught the exception to retrieve the result
        return (Item) found.getFoundObject();
    }
    return null; // not found
}

// Custom exception used to control flow
class FoundException extends RuntimeException {
    private Object foundObject;
    public FoundException(Object obj) { this.foundObject = obj; }
    public Object getFoundObject() { return foundObject; }
}
```

In this example, the author is throwing `FoundException` to break out of the `forEach` lambda when the item is found, and then catching it to get the result.

Improved Solution – Normal Control Flow: Use regular control flow constructs to handle logic. For instance, a simple loop that returns the item when found (or uses streams with a filter) is clearer and doesn't abuse exceptions:

```
public Item findItem(List<Item> items, String id) {
    for (Item item : items) {
        if (item.getId().equals(id)) {
            return item;  // normal flow: return the found item
        }
    }
    return null; // not found
}

// Alternatively, using streams:
public Item findItemStream(List<Item> items, String id) {
    return items.stream()
                .filter(item -> item.getId().equals(id))
                .findFirst()
                .orElse(null);
}
```

Why It's Better: The improved code uses standard loop/stream mechanisms to return the result without exceptions. This makes the logic more readable and efficient. There's no need for a special `FoundException` or try-catch for normal operations. Using exceptions for flow control can clutter logs with stack traces that aren't really errors, and can degrade performance. In microservices, where performance and clear logging are important, avoiding this anti-pattern leads to cleaner, faster, and more maintainable code.

8. Shared Mutable State (Global State)

Microservices are typically designed to be stateless or manage state carefully. Using shared mutable state — for example, static variables, global singletons holding data, or caches that are not properly synchronized — is an anti-pattern. It can lead to unpredictable behavior in a multi-threaded environment and issues when scaling the service (because each instance has its own in-memory state that may diverge). It also makes testing harder since tests might interfere via shared state.

Anti-Pattern Example – Static Mutable State: In this example, a static `Map` is used to cache user sessions. The map is mutable and shared across all threads in the service. If multiple threads modify it, it can cause race conditions unless synchronized. Additionally, in a microservice cluster, each instance has its own map (not shared across instances), leading to inconsistent state if the service is scaled or restarted:

```
public class SessionManager {
    // Anti-pattern: static mutable state shared globally
    private static final Map<String, SessionData> sessions = new HashMap<>();

    public void addSession(String userId, SessionData data) {
        sessions.put(userId, data);  // no synchronization, potential race
condition
    }

    public SessionData getSession(String userId) {
        return sessions.get(userId); // returns null if not in this instance's
map
    }
}
```

Here, if two threads call `addSession` at the same time, `HashMap` (not thread-safe) can behave unpredictably. Also, if in a cloud environment with 3 instances of this microservice, a user might hit instance A to create a session and instance B to retrieve it, and B won't find it because the state isn't shared.

Improved Solution – Avoid Global Mutable State: There are a few strategies to fix this: - **Make the state instance-bound and thread-safe:** Use proper synchronization or concurrent collections if truly needed in one instance. - **Externalize the state:** e.g., use a distributed cache or database that all instances share. - **Design for statelessness:** e.g., issue a token to the client that encodes session info (JWT) or use a dedicated session service.

Below, we show using an external store (pretend `SessionRepository` abstracts a DB or cache). The static map is removed; each session write/read goes through a thread-safe repository:

```
public class SessionManager {
    private final SessionRepository repo; // e.g., an interface to Redis or DB

    public SessionManager(SessionRepository repo) {
        this.repo = repo;
    }

    public void addSession(String userId, SessionData data) {
        repo.save(userId, data);  // store in external shared storage
    }

    public SessionData getSession(String userId) {
        return repo.find(userId); // retrieve from shared storage
    }
}
```

Alternatively, if sessions must be in-memory for performance, use a thread-safe structure and consider using a distributed cache:

```
// Using a thread-safe map (and possibly a distributed cache like Hazelcast)
private static final ConcurrentMap<String, SessionData> sessions = new
ConcurrentHashMap<>();
```

But even a ConcurrentHashMap on each instance will not share data across instances, so externalizing is key for truly shared state.

Why It's Better: Removing shared mutable state reduces bugs from concurrency and inconsistency. In the improved approach, session data is not kept in a static structure that's hidden in the application memory. By using a `SessionRepository` (which could be backed by an external system), all microservice instances see a consistent view of sessions. This also allows scaling the service horizontally without losing session data. If using in-memory caches, using thread-safe collections prevents concurrent modification errors. Overall, avoiding global mutable state makes microservices more robust, easier to scale, and simpler to reason about (since each instance doesn't carry hidden state that can change unpredictably).

9. Incorrect Singleton Implementation (Double-Checked Locking Issue)

The Singleton pattern is sometimes needed (for example, a single instance of a heavy object like a config manager). However, a common anti-pattern in Java is implementing singleton initialization with the flawed double-checked locking idiom (without proper precautions). Prior to Java 5, and even after if done incorrectly, double-checked locking can result in race conditions where two threads create two instances, or a thread sees a half-initialized object. This is a concurrency anti-pattern that can badly affect a microservice under load.

Anti-Pattern Example – Broken Double-Checked Locking: In this code, the intention is to only create one instance of `ConfigLoader`. The code checks `if (instance == null)` then synchronizes and checks again, which is the double-checked locking pattern. However, if `instance` is not declared `volatile`, this pattern can fail: one thread may see a non-null `instance` that is not fully constructed yet. This can lead to using an uninitialized object:

```
public class ConfigLoader {
    private static ConfigLoader instance;
    private Properties config;

    private ConfigLoader() {
        // Load configuration from file or service
        config = loadConfig();
    }

    public static ConfigLoader getInstance() {
```

```
        if (instance == null) {                     // first check (not
synchronized)
            synchronized(ConfigLoader.class) {
                if (instance == null) {             // second check (in lock)
                    instance = new ConfigLoader();  // Anti-pattern: not using
volatile
                }
            }
        }
        return instance;
    }

    public String getValue(String key) {
        return config.getProperty(key);
    }
}
```

This code might appear to work in testing, but under heavy concurrency, thread A could create the object while thread B, seeing `instance` not null (but possibly not fully constructed yet due to reordering), returns a partially initialized `config`.

Improved Solution – Thread-Safe Singleton: There are multiple correct ways to create singletons: - Use an initialization-on-demand holder (static inner class) or an `enum` singleton (which is simplest and inherently thread-safe). - Or use the double-checked locking *with* a `volatile` variable in Java 5+.

Below is an improved version using the initialization-on-demand holder idiom, which is thread-safe without explicit synchronization:

```
public class ConfigLoader {
    private Properties config;
    private ConfigLoader() {
        config = loadConfig();
    }
    // Static inner class responsible for holding the single instance
    private static class Holder {
        static final ConfigLoader INSTANCE = new ConfigLoader();
    }
    public static ConfigLoader getInstance() {
        return Holder.INSTANCE;
    }
    public String getValue(String key) {
        return config.getProperty(key);
    }
}
```

Alternatively, using an `enum`:

```
public enum ConfigLoaderSingleton {
    INSTANCE;
    private Properties config;
    private ConfigLoaderSingleton() {
        config = loadConfig();
    }
    public String getValue(String key) {
        return config.getProperty(key);
    }
}
```

If sticking with double-checked locking, declaring `instance` as `private static volatile ConfigLoader instance;` would be required to prevent caching issues.

Why It's Better: The holder-class or enum approaches leverage classloading guarantees to ensure thread-safe lazy initialization without complicated locking. The improved solutions ensure only one instance will ever be created in any thread-safe manner. This prevents subtle bugs that could emerge under high concurrency (which microservices might face). Using these patterns avoids the complexity and pitfalls of manual double-checked locking. As a result, the configuration (or any singleton resource) is initialized safely exactly once, and all threads see a fully constructed object.

10. Poor Thread Management (Creating Unmanaged Threads)

Manually creating and managing threads in application code is usually an anti-pattern in a managed environment (like a Java EE container or Spring Boot application). It can lead to threads that are not properly pooled or lifecycle-managed, causing resource leaks or unpredictable performance. In microservices, which often run in application servers or frameworks, you should leverage provided thread pools or executors. Spawning raw threads can interfere with container-managed resources and makes controlling concurrency harder (e.g., you might accidentally create too many threads).

Anti-Pattern Example – Uncontrolled Thread Creation: In this code, a new thread is started for a task. There's no reuse (every request might spawn a new thread) and no way to shut it down gracefully. If this is done repeatedly, it could create an unbounded number of threads, exhausting system resources:

```
public class ReportGenerator {
    public void generateReportAsync(Runnable task) {
        // Anti-pattern: creating a new thread for each task
        Thread t = new Thread(task);
        t.setDaemon(true);
        t.start();
    }
}

// Usage:
```

```
reportGenerator.generateReportAsync(() -> {
    // do heavy report generation
});
```

This bypasses any thread pool. Also, if the microservice needs to shut down, these threads might not finish or might be abruptly stopped if daemon threads.

Improved Solution – Use Thread Pools or Async Framework: Instead of `new Thread()`, use a thread pool or high-level async mechanism. For example, use `ExecutorService` or if using Spring, an `@Async` method that uses a configured thread pool. This allows reusing threads and controlling their lifecycle:

```
public class ReportGenerator {
    private final ExecutorService executor;
    public ReportGenerator(ExecutorService executor) {
        this.executor = executor; // injected or created with a fixed pool
    }
    public void generateReportAsync(Runnable task) {
        executor.submit(task); // uses the thread pool
    }
}

// Usage, perhaps configured with a fixed thread pool of size N:
ExecutorService pool = Executors.newFixedThreadPool(10);
ReportGenerator gen = new ReportGenerator(pool);
gen.generateReportAsync(() -> {
    // do heavy report generation
});
```

In a Spring Boot microservice, one could annotate the method with `@Async` and configure an `AsyncExecutor` bean, avoiding manual thread handling entirely:

```
@Async
public CompletableFuture<Report> generateReportAsync(Data data) {
    // ... runs in a thread from Spring's thread pool
}
```

Why It's Better: Using a thread pool or container-managed async tasks ensures that you don't oversubscribe the system with threads. Thread pools can limit and reuse threads, which is more efficient and safer. The improved approach also makes it easier to handle application shutdown (pools can be shut down gracefully). In microservices, this is critical; for example, uncontrolled threads might continue running during redeploy or prevent the service from stopping cleanly. By managing threads properly, the service remains reliable and resource-conscious, and you avoid hard-to-debug issues like thread leaks or race conditions from too many concurrent threads.

11. Ignoring Thread Interruption

In Java, threads can be interrupted (for example, to signal them to stop work). A common anti-pattern is catching `InterruptedException` and doing nothing, or not restoring the interrupt status. This can happen in microservices that perform background tasks or use concurrency utilities. Ignoring interruptions means your thread may not respond to cancellation requests (like shutting down the service), leading to threads that keep running when they should stop.

Anti-Pattern Example – Swallowing InterruptedException: In this code, the thread sleeps for a bit between retries. If the thread is interrupted (perhaps the service is shutting down), the catch just logs and continues, effectively ignoring the interruption and continuing the loop:

```
public void retryOperation() {
    for (int i = 0; i < 5; i++) {
        try {
            performOperation();
            break; // success
        } catch (TransientFailureException e) {
            System.err.println("Retrying after failure: " + e.getMessage());
            try {
                Thread.sleep(1000);  // pause before retry
            } catch (InterruptedException ie) {
                // Anti-pattern: interruption ignored
                System.err.println("Sleep interrupted, but continuing...");
                // (not rethrowing or handling interruption properly)
            }
        }
    }
}
```

In this scenario, if the thread was interrupted (perhaps by an ExecutorService shutdown), the code just prints and goes on to retry, instead of stopping the retry loop.

Improved Solution – Handle Interruption Properly: When catching `InterruptedException`, either propagate it (if you want higher-level logic to handle it) or at least re-set the thread's interrupt flag. A common pattern is to `Thread.currentThread().interrupt()` in the catch, or throw an `InterruptedException` up. Here we choose to break out of the loop when interrupted:

```
public void retryOperation() throws InterruptedException {
    for (int i = 0; i < 5; i++) {
        try {
            performOperation();
            break; // success, exit loop
        } catch (TransientFailureException e) {
            log.warn("Operation failed, will retry... (" + e.getMessage() +
```

```
")");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                // Proper handling: restore interrupt status and stop further
processing
                Thread.currentThread().interrupt();
                log.info("Thread interrupted, aborting retries.");
                break; // break out of loop if interrupted
            }
        }
    }
}
```

If this method is in a runnable, the `InterruptedException` could be thrown to let the executor or caller know the thread was interrupted.

Why It's Better: Properly handling interrupts ensures that your threads can respond to shutdown signals or cancellation. In the improved code, if an interrupt happens, we restore the interrupt status and break out of the retry loop, preventing further work. This is important in microservices when shutting down gracefully — threads listening for interrupts should stop promptly. Ignoring interrupts could delay shutdown or waste resources doing work that is no longer needed. By respecting the interrupt, we make our service more responsive to control signals and avoid potential issues like stuck threads or prolonged downtime during redeployments.

12. Not Closing Resources (Resource Leaks)

Forgetting to release resources such as database connections, file streams, or network sockets is a serious anti-pattern. In a long-running microservice, even small resource leaks can accumulate and eventually exhaust resources (file handles, connection pool, memory), causing the service to degrade or crash. Proper try-with-resources or finally blocks to close resources are essential.

Anti-Pattern Example – Resource Leak: In this code, a file is opened but never closed. If this method is called repeatedly, file handles will remain open until the process ends. In a microservice running continuously, this could eventually hit the open files limit or lock the file:

```
public String readFirstLine(String path) throws IOException {
    // Anti-pattern: not closing the FileReader/BufferedReader
    FileReader fr = new FileReader(path);
    BufferedReader br = new BufferedReader(fr);
    return br.readLine();  // after this, the file remains open
}
```

Similarly, neglecting to close database `ResultSet` / `Statement` / `Connection` or not returning connections to the pool is a common resource leak.

Improved Solution – Use Try-with-Resources (Auto-Close): Using Java's try-with-resources or finally blocks to ensure closure of resources guarantees that they are released. The code below uses try-with-resources, which will automatically close the reader after use, even if an exception occurs:

```
public String readFirstLine(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
    // FileReader is wrapped by BufferedReader and will be closed automatically
here
}
```

For database operations:

```
public User getUser(int id) throws SQLException {
    String sql = "SELECT * FROM users WHERE id=?";
    try (Connection conn = dataSource.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql)) {
         stmt.setInt(1, id);
         try (ResultSet rs = stmt.executeQuery()) {
             if (rs.next()) {
                 return mapUser(rs);
             }
         }
    } // conn, stmt, rs all auto-closed here
    return null;
}
```

Why It's Better: The improved code ensures that resources are freed promptly. Using try-with-resources leads to cleaner code that is less error-prone than manually closing in finally blocks. In a microservice, which may handle thousands of requests, each opening files or connections, proper resource management prevents exhaustion of file descriptors, memory, or DB connections. Leaking resources can lead to performance degradation over time (e.g., connection pool exhaustion causing requests to hang). By closing everything properly, we keep the service stable and avoid mysterious issues that only appear under load or after long uptimes.

13. Using System.out for Logging

Using `System.out.println` or `System.err` for logging in a server application is an anti-pattern. It bypasses the logging framework, cannot be easily managed (no log levels, no rotation), and mixes with other console output. In microservices, proper logging (with a framework like Log4j, SLF4J, etc.) is crucial for debugging and monitoring. System.out prints are also synchronous writes to console/STDOUT, which can be slow and impact performance if done excessively.

Anti-Pattern Example – Println Logging: The code logs an event by printing to standard output. This provides no severity level and cannot be filtered or turned off easily in production. It also doesn't include structured context like timestamps unless the environment captures stdout and adds it:

```
public void updateOrderStatus(Order order, String status) {
    // Anti-pattern: using System.out for logging
    System.out.println("Updating order " + order.getId() + " status to " +
status);
    order.setStatus(status);
    orderRepository.save(order);
    System.out.println("Order " + order.getId() + " updated.");
}
```

Improved Solution – Use a Logging Framework: The corrected version uses a logger (for example, SLF4J with Logback). It logs at an appropriate level (info, debug, etc.), which can be configured. The logger automatically includes timestamps and can be directed to files or monitoring systems:

```
private static final Logger log = LoggerFactory.getLogger(OrderService.class);

public void updateOrderStatus(Order order, String status) {
    log.info("Updating order {} status to {}", order.getId(), status);
    order.setStatus(status);
    orderRepository.save(order);
    log.debug("Order {} updated successfully.", order.getId());
}
```

Why It's Better: A proper logging framework allows controlling the output (format, destination, level). For instance, in production you might log only warnings/errors to avoid noise, whereas in development, info/debug are enabled. The improved code uses placeholders `{}` which is efficient (avoids string concatenation when the log level is disabled) and outputs structured messages. In microservices, logs are often aggregated; using a consistent format and framework means your logs can be parsed by tools. `System.out.println` lines are harder to manage and could even be lost if the process stdout isn't captured. Additionally, flooding System.out can degrade performance. By using a logger, we gain flexibility and maintainability for our logging strategy.

14. Logging Sensitive Information

While logging is important, logging sensitive data (such as passwords, credit card numbers, personal details, or secret tokens) is an anti-pattern. It can lead to security breaches if logs are accessible, and it violates privacy. In a microservices ecosystem, logs often go to centralized systems; any sensitive info logged can spread to many places and become difficult to scrub.

Anti-Pattern Example – Logging Secrets: This example logs a password and credit card number, which is highly sensitive. Even though it might be for debugging, it should never be in logs, especially in plain text:

```
public boolean authenticate(String user, String password) {
    // Anti-pattern: logging sensitive data (password)
    log.debug("Authenticating user: {} with password: {}", user, password);
    boolean success = authService.verify(user, password);
    log.debug("Auth result for card {}: {}", userCreditCard, success);
    return success;
}
```

Improved Solution – Sanitize or Avoid Sensitive Data: Remove or mask sensitive information from logs. Only log non-sensitive context. For example, don't log the password at all, and partially mask credit card numbers if needed:

```
public boolean authenticate(String user, String password) {
    log.debug("Authenticating user: {}", user);
    boolean success = authService.verify(user, password);
    log.debug("Auth result for user {}: {}", user, success);
    return success;
}

// If logging something like a credit card, mask it:
public void chargeCard(String cardNumber) {
    String masked = maskCard(cardNumber); // e.g., "************1234"
    log.info("Charging credit card {}", masked);
    // proceed with charging
}
```

In the above, `maskCard` would leave only last 4 digits visible, for instance.

Why It's Better: By not logging sensitive info, we protect user data and system security. If an error occurs during authentication or payment, we can log that an error happened without exposing secrets. This is critical for compliance (like GDPR, PCI-DSS for credit cards). In a microservice architecture, logs might be widely distributed (in log files, monitoring dashboards, etc.), so one slip-up can leak data in multiple places. The improved practice ensures that even if logs are accessed, they won't contain critical secrets. It maintains user trust and adheres to security best practices.

15. N+1 Query Problem (Inefficient Data Access)

The N+1 queries anti-pattern occurs when code retrieves a list of entities and then, in a loop, fetches additional data for each entity with a separate query. This results in 1 (for the initial list) + N (for each element) database queries, which is extremely inefficient and can slow down the service. In microservices that often use ORMs (like JPA/Hibernate) or make multiple service calls, this anti-pattern can cause performance issues and high load on the database or network.

Anti-Pattern Example – N+1 Database Queries: Suppose we need to get all orders with their items. The code first fetches orders, then for each order, it queries the items. This leads to many queries (one per order):

```
// Anti-pattern: N+1 query issue
List<Order> orders = orderRepository.findAll();  // Query 1: fetch all orders
for (Order order : orders) {
    List<Item> items = itemRepository.findByOrderId(order.getId()); // Query N:
one per order
    order.setItems(items);
}
return orders;
```

If there are 100 orders, this will execute 1 + 100 = 101 queries, which is slow and unnecessary.

Improved Solution – Batch or Join Fetch: Use a single query with a join or an ORM feature to fetch related data in one go, or batch the fetches. For instance, using JPQL or criteria to fetch orders with their items in one query, or configuring the ORM to use eager fetching appropriately:

```
// Solution 1: Use a JOIN fetch in query (if using JPA/Hibernate)
List<Order> orders = orderRepository.findAllWithItems();
// e.g., @Query("SELECT o FROM Order o JOIN FETCH o.items") in repository

// Solution 2: Batch load items by collecting IDs
List<Order> orders = orderRepository.findAll();
List<Long> orderIds =
orders.stream().map(Order::getId).collect(Collectors.toList());
List<Item> allItems = itemRepository.findAllByOrderIdIn(orderIds);
// one query to get items for all orders
Map<Long, List<Item>> itemsByOrder =
allItems.stream().collect(Collectors.groupingBy(Item::getOrderId));
for (Order order : orders) {
    order.setItems(itemsByOrder.getOrDefault(order.getId(),
Collections.emptyList()));
}
return orders;
```

In Solution 2, we reduced N+1 queries to just 2 queries: one for orders and one for all items needed.

Why It's Better: The improved approach drastically reduces the number of database calls. In microservices handling many requests, N+1 queries can cause high latency and load (e.g., hundreds of DB calls per request). By fetching in batches or using joins, we retrieve all needed data with minimal queries, improving performance. This not only speeds up the response but also reduces load on the database. Advanced ORMs have ways to solve this (like `@OneToMany(fetch = FetchType.EAGER)` with caution, or join fetch

queries). The key is being aware of the N+1 pattern and restructuring code or queries to avoid it. Efficient data access is crucial for scalable microservices.

16. Shared Database Anti-Pattern (Tight Coupling via Database)

In microservice architectures, each service ideally owns its data store. The shared database anti-pattern occurs when multiple services directly connect to the same database (or schema), or one service queries another service's tables. This creates tight coupling: a change in the database schema affects all services, and services become interdependent. It breaks the modularity of microservices and can lead to deployment and data integrity problems.

Anti-Pattern Example – Direct Table Access Across Services: Consider two services: `OrderService` and `InventoryService`. In the bad example, `InventoryService` directly reads the `orders` table from the OrderService's database to get order info. This bypasses the OrderService's API and couples the two at the data level:

```
// In InventoryService, directly using OrderService's database table:
public int countRecentOrdersForProduct(String productId) throws SQLException {
    Connection conn = DriverManager.getConnection("jdbc:postgresql://orders-db:
5432/orders", "user", "pass");
    String sql = "SELECT COUNT(*) FROM orders o WHERE o.product_id = ?";
    try (PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setString(1, productId);
        try (ResultSet rs = ps.executeQuery()) {
            rs.next();
            return rs.getInt(1);
        }
    }
}
```

In this scenario, if the `orders` table schema changes or the OrderService's DB is down, `InventoryService` breaks. Also, `InventoryService` now needs credentials to `orders` DB, creating a security concern.

Improved Solution – Use Service API or Messaging: The InventoryService should request needed info via the OrderService's API (or perhaps listen on an event stream). This keeps the boundary intact. For example, InventoryService calls a REST endpoint on OrderService to get the count, or OrderService could emit an event when an order is placed and InventoryService tracks counts. Here's a simple API call approach:

```
// In InventoryService, using OrderService API instead of direct DB access
public int countRecentOrdersForProduct(String productId) {
    // Assume OrderService has an API: GET /orders/count?productId=XYZ
    String url = "http://orders-service/orders/count?productId=" + productId;
    HttpResponse<String> response =
httpClient.send(HttpRequest.newBuilder(URI.create(url)).build(),
```

```
                                                    BodyHandlers.ofString());
    if (response.statusCode() == 200) {
        return Integer.parseInt(response.body());
    } else {
        throw new RuntimeException("Failed to fetch order count");
    }
}
```

(Pseudocode for calling another service's REST API.)

Alternatively, use an event-driven approach: OrderService publishes an event "OrderCreated" with product info, and InventoryService subscribes and updates its own data store.

Why It's Better: By using the service's API or events, we preserve loose coupling. Each microservice owns its data and others access it only through well-defined contracts. This means services can change their internal implementation or schema without breaking others, as long as the contract is maintained. It also improves security (services only expose necessary data, and credentials to one DB stay with that service) and allows independent scaling. The improved approach may have slight overhead (calls between services or managing an event pipeline), but it ensures a more robust, maintainable architecture. In summary, microservices should talk to each other over the network, not via each other's databases.

17. Missing Timeouts in External Calls

When a microservice calls an external resource (another service, a database, an API), not specifying timeouts is an anti-pattern. By default, many HTTP clients or DB drivers will wait indefinitely for a response. In a distributed system, failing to time out can lead to threads hanging forever, resource exhaustion, and cascading failures (if one service becomes unresponsive and others pile up waiting on it). Always use timeouts to bound how long you wait for a response.

Anti-Pattern Example – No Timeout on HTTP Call: In this example, the service calls an external REST API using a default HTTP client configuration. If the external service hangs or is very slow, this call could block indefinitely, holding up the thread:

```
public String fetchUserData(String userId) {
    // Anti-pattern: no timeout specified for HTTP request
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("http://external-service/users/" + userId))
            .build();
    try {
        HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
        return response.body();
    } catch (IOException | InterruptedException e) {
        throw new RuntimeException("Call failed", e);
```

```
    }
}
```

Here, `HttpClient.send` will use default settings – by default Java 11 HttpClient uses no explicit timeout (it may rely on TCP timeouts which are very long). Similarly, a JDBC call without query timeout could hang if the DB doesn't respond.

Improved Solution – Configure Timeouts and Fallbacks: Always set a reasonable timeout on external calls. For HTTP, use a timeout on the client or request. For example:

```
public String fetchUserData(String userId) {
    HttpClient client = HttpClient.newBuilder()
            .connectTimeout(Duration.ofSeconds(2))
            .build();
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("http://external-service/users/" + userId))
            .timeout(Duration.ofSeconds(3))  // 3 seconds timeout for the
response
            .build();
    try {
        HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
        if (response.statusCode() == 200) {
            return response.body();
        } else {
            // handle non-200 responses
            throw new RuntimeException("Got error status: " +
response.statusCode());
        }
    } catch (IOException | InterruptedException e) {
        log.error("External call failed or timed out", e);
        // Fallback or propagate a specific exception
        throw new ExternalServiceException("User data service unavailable", e);
    }
}
```

For a database query, one could use JDBC's `Statement.setQueryTimeout(...)` or configure in the connection pool.

Why It's Better: With timeouts, if the external service is unresponsive, our thread will only wait a fixed amount of time (e.g., 3 seconds) before giving up. This prevents piling up threads waiting forever, which can crash a service under load. The improved code also considers handling of non-200 HTTP responses and exceptions, making the service more robust. In microservices, it's common to have timeout strategies to maintain overall responsiveness (e.g., one service might degrade functionality if another is slow, rather than hang). By using timeouts and handling failures gracefully (possibly with fallback logic or meaningful error messages), we ensure that one bad component doesn't bring down the entire system.

18. No Retry or Circuit Breaker (Lack of Resilience)

In distributed systems, calls between services or to databases might fail intermittently. Simply propagating the failure or constantly retrying without limits is an anti-pattern. A lack of resilience mechanisms like retries (with backoff) or circuit breakers can make the overall system fragile. For example, if a service is temporarily down, clients should not hammer it non-stop (that could worsen the issue). Not implementing these patterns means microservices can easily get into cascades of failures.

Anti-Pattern Example – No Retry or Fallback: The code calls another service but if it fails, it just immediately returns error. There is no attempt to retry a transient failure, and no circuit breaker to stop repeated failing calls. If this call is part of a larger operation, a brief glitch can cause the entire operation to fail without a second attempt:

```
public Order getOrderWithDetails(String orderId) {
    // Call to another microservice for order details
    try {
        String orderJson = httpClient.get("http://order-service/orders/" +
orderId);
        return parseOrder(orderJson);
    } catch (IOException e) {
        // Anti-pattern: immediately give up on failure
        log.error("Failed to get order details", e);
        return null;
    }
}
```

This approach may be acceptable for some failures, but if the error was a temporary network blip, we lost the chance to recover. Also, if order-service is down, this method will be called every time without any backoff, possibly flooding the network or logs with errors.

Improved Solution – Implement Retry and Circuit Breaker: Use a limited retry strategy for transient errors, and a circuit breaker to avoid constant failing calls. For example, use a library like Resilience4j or Hystrix (if available), or implement simple logic: - Retry: attempt the call a few times with delays. - Circuit Breaker: if failures exceed a threshold, skip calling for a cooldown period (immediately fail or use cached/default response).

Here's a conceptual example using Resilience4j-like pseudocode:

```
public Order getOrderWithDetails(String orderId) throws
ExternalServiceException {
    // Configure a circuit breaker and retry (could be done via annotations or
builder)
    CircuitBreaker cb = CircuitBreaker.ofDefaults("orderServiceCB");
    Retry retry = Retry.ofDefaults("orderServiceRetry");
```

```
    // The supplier is the code that calls the external service
    Supplier<Order> orderSupplier = () -> {
        try {
            String orderJson = httpClient.get("http://order-service/orders/" +
orderId);
            return parseOrder(orderJson);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    };

    // Decorate the supplier with resilience patterns
    Supplier<Order> resilientSupplier = CircuitBreaker.decorateSupplier(cb,
                                        Retry.decorateSupplier(retry,
orderSupplier));
    try {
        return resilientSupplier.get();
    } catch (CallNotPermittedException e) {
        // circuit breaker open
        log.warn("Order service circuit breaker open, returning default");
        return new Order(orderId, Collections.emptyList()); // default fallback
    } catch (Exception e) {
        throw new ExternalServiceException("Failed to get order details after
retries", e);
    }
}
```

(In real code, you would configure `CircuitBreaker` *and* `Retry` *parameters like max attempts, wait duration, failure rates, etc. Here it's simplified.)*

If not using libraries, a simple manual retry might look like:

```
for (int attempt = 1; attempt <= 3; attempt++) {
    try {
        String orderJson = httpClient.get(url);
        return parseOrder(orderJson);
    } catch (IOException e) {
        log.warn("Attempt " + attempt + " failed to fetch order " + orderId);
        if (attempt == 3) throw e;
        Thread.sleep(100); // small backoff before retry
    }
}
```

And a crude circuit breaker could be a timestamp or counter to skip calls if too many recent failures.

Why It's Better: The improved approach makes the service more resilient to temporary issues. A retry mechanism can recover from intermittent failures without exposing an error to the user. A circuit breaker prevents continuous attempts to a down service, allowing it time to recover and preventing resource waste (and avalanche of logs). In microservices, these patterns are vital to achieve stability: they localize failures and give flexibility in handling them (like returning a default value or cached data when a service is down). Without them, a single service outage can easily cascade into others as they pile up failures or hang (if no timeouts, as mentioned earlier). With them, the system can degrade gracefully.

19. Overly Chatty Communication (Excessive Remote Calls)

A chatty microservice is one that makes numerous small calls to other services (or database) to accomplish a single task. This is an anti-pattern because remote calls have overhead (network latency, serialization) and making too many can dramatically slow down the system and increase load. Often this happens when services are too fine-grained or when one service repeatedly requests data item-by-item instead of in bulk. Advanced developers should look to minimize round trips.

Anti-Pattern Example – Chatty Client Calls: The example below retrieves details for a list of product IDs by calling a product service for each ID in a loop. If there are 50 products, it will make 50 synchronous calls to another service sequentially, which is very slow:

```
public List<Product> getProductsDetails(List<String> productIds) {
    List<Product> products = new ArrayList<>();
    for (String pid : productIds) {
        // Anti-pattern: calling remote service in a tight loop (N calls)
        String url = "http://product-service/products/" + pid;
        String json = httpClient.get(url);
        products.add(parseProduct(json));
    }
    return products;
}
```

If each call takes 50ms, 50 calls = 2.5 seconds just in latency, not counting processing. And if done frequently, this also puts load on product-service repeatedly.

Improved Solution – Batch or Aggregate Calls: Redesign to reduce the number of calls. Perhaps the product service provides a batch API to fetch multiple products in one request, or fetch all needed data in one go. Alternatively, caching can help if the same data is repeatedly fetched. Here's a solution using a batch API:

```
public List<Product> getProductsDetails(List<String> productIds) {
    // Improved: call a batch API with all IDs or use query parameters
    String idsParam = String.join(",", productIds);
    String url = "http://product-service/products?ids=" + idsParam;
    String jsonResponse = httpClient.get(url);
```

```
    return parseProductsList(jsonResponse);
}
```

Now one call returns all 50 products. If such API is not available, another approach is to fetch in parallel (if the client and network can handle it) rather than sequentially, using asynchronous calls to overlap latency. Or use caching: if these product details are frequently needed, cache them so you don't call every time.

Why It's Better: Reducing chatty communication improves performance and scalability. The batch call consolidates overhead — we pay the latency cost once instead of 50 times. It also reduces network chatter and the load on the product service (which can fetch all requested products with one database query instead of 50 separate queries). In microservices design, one should strive for coarse-grained APIs that fulfill useful chunks of data, rather than extremely fine-grained calls. When multiple calls are unavoidable (e.g., complex workflows), consider asynchronous parallel calls or caching results. Overall, the improved pattern leads to faster response times and less inter-service traffic, which helps the system scale and be more resilient to network issues.

20. Big Ball of Mud (Lack of Separation of Concerns)

A "big ball of mud" refers to code that has no clear structure or separation — everything is tangled together. In a microservice, this anti-pattern might manifest as a single class or method doing too many things: handling HTTP, business logic, database access, etc., all in one place. This violates separation of concerns and single responsibility principle, making the code hard to maintain or test. Advanced developers should structure code into layers or components (even within a microservice) for clarity.

Anti-Pattern Example – Monolithic Method: Below is an example of a single method that is handling validation, business logic, and data persistence all at once. It's long and tries to do everything, which makes it error-prone and difficult to modify or reuse parts of it:

```
// Anti-pattern: one method doing too much
public Order placeOrder(String userId, List<Item> items) {
    // Validate input
    if (userId == null || items.isEmpty()) {
        throw new IllegalArgumentException("Invalid order");
    }
    // Calculate total
    double total = 0;
    for (Item item : items) {
        total += item.getPrice();
    }
    // Business logic: e.g., apply discount
    if (total > 1000) {
        total *= 0.9; // 10% discount for large orders
    }
    // Persist order to database
    Connection conn = dataSource.getConnection();
    try (PreparedStatement ps = conn.prepareStatement("INSERT INTO orders ...",
```

```
Statement.RETURN_GENERATED_KEYS)) {
        // set fields...
        ps.executeUpdate();
        // ... insert order items
    }
    // Call external service to notify
    httpClient.post("http://notification-service/notifyOrder", /* order info
*/ );

    // Return result
    Order order = new Order(/*...*/);
    order.setTotal(total);
    return order;
}
```

This method mixes validation, calculation, database code, and an external service call. Testing it requires a database and an HTTP endpoint unless you heavily mock things, and any change (like a new discount rule) means touching this large method.

Improved Solution – Layer and Refactor: Split responsibilities into separate methods or classes: e.g., a `OrderValidator`, an `OrderCalculator`, a `OrderRepository`, and a `NotificationClient`. The `placeOrder` method in a service class then orchestrates these. This not only respects single responsibility, but also makes each piece testable in isolation:

```
public class OrderService {
    private OrderValidator validator;
    private OrderCalculator calculator;
    private OrderRepository orderRepo;
    private NotificationClient notificationClient;

    public Order placeOrder(String userId, List<Item> items) {
        validator.validateOrder(userId, items);
        double total = calculator.calculateTotal(items);
        Order order = new Order(userId, items, total);
        orderRepo.save(order);                    // persist to DB
        notificationClient.notifyOrderPlaced(order); // notify external service
        return order;
    }
}

// Each component does one thing:
public class OrderValidator {
    public void validateOrder(String userId, List<Item> items) {
        if (userId == null || items.isEmpty()) {
            throw new IllegalArgumentException("Invalid order");
        }
```

```
    }
}
public class OrderCalculator {
    public double calculateTotal(List<Item> items) {
        double total = items.stream().mapToDouble(Item::getPrice).sum();
        if (total > 1000) {
            total *= 0.9;
        }
        return total;
    }
}
public interface OrderRepository {
    Order save(Order order);
}
public class NotificationClient {
    public void notifyOrderPlaced(Order order) {
        httpClient.post("http://notification-service/notifyOrder",
order.toJson());
    }
}
```

Why It's Better: The improved code separates concerns: validation logic is not mixed with persistence or external calls. Each part can change independently. For example, changing how discounts are calculated only affects `OrderCalculator`. It also makes unit testing straightforward: you can test the calculator or validator alone without needing a database or HTTP server. The `OrderService.placeOrder` method is now an orchestrator of these components; it's shorter and clearly shows the high-level flow. In microservices, even though the service might be small, internal structure matters for maintainability. A well-structured microservice codebase will be easier to extend and less prone to bugs than a big ball of mud where everything is entangled. This leads to more robust services that are easier for teams to work on and evolve over time.

Conclusion:

In this guide, we covered a range of Java anti-patterns with particular relevance to microservices and modern service-oriented architecture. Each anti-pattern can undermine the reliability, scalability, or maintainability of your service. By recognizing and addressing these issues — from configuration and error handling to concurrency and integration practices — advanced developers can ensure their microservices remain robust and easy to manage. The examples provided show both bad practices and recommended solutions, offering a clear illustration of how to improve code quality and system design. Remember that writing clean, resilient code is as important as the service architecture itself in achieving a successful microservices ecosystem. Happy coding!
