1. One Codebase

While less of a Java-specific concept, this factor generally refers to getting to a single code base managed in source control or a set of repositories from a common root. Getting to a single codebase makes it cleaner to build and push any number of immutable releases across various environments. The best example of violating this is when your app is composed of a dozen or more code repositories. While using one code repository to produce multiple applications can be workable, the goal is a 1:1 relationship between apps and repos. Operating from one codebase can be done but is not without its own challenges. Sometimes one application per repository is the simplest thing that works for a team or organization.

2. Dependency Management

Most Java (and Groovy) developers can take advantage of facilities like Maven (and Gradle), which provide the means to declare the dependencies your app requires for proper build and execution. The idea is to allow developers to declare dependencies and let the tool ensure those dependencies are satisfied and packaged into a single binary deployment artifact. Plugins like Maven Shade or Spring Boot enable you to bundle your application and its dependencies into a single “uberjar” or “fat jar” and thus provide the means to isolate those dependencies.

Figure 1 is a portion of an example Spring Boot application Maven build file, pom.xml. This shows the dependency declarations as specified by the developer.

Figure 1: A portion of POM.xml showing application dependencies

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository
    -->
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zipkin</artifactId>
    </dependency>
```

Figure 2 is a portion of listed dependencies within the same application, showing JARs bundled into the application's uberjar, which isolates those dependencies from variations in the underlying environment. The application will rely upon these dependencies rather than potentially conflicting libraries present in the deployment target.

Figure 2: A portion of mvn dependency:tree for a sample application

```
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building quote-service 0.0.1-SNAPSHOT
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.10:tree (default-cli) @quote-service ---
[INFO] com.example:quote-service:jar:0.0.1-SNAPSHOT
[INFO] +-
org.springframework.cloud:spring-cloud-starterconfig:jar:1.1.3.RELEASE:compil
e
[INFO] | +-
org.springframework.cloud:spring-cloud-starter:jar:1.1.1.RELEASE:compile
[INFO] | | +-
org.springframework.cloud:spring-cloud-context:jar:1.1.1.RELEASE:compile
[INFO] | | | \-
org.springframework.security:springsecurity-crypto:jar:4.0.4.RELEASE:compile
[INFO] | | +-
org.springframework.cloud:spring-cloud-commons:jar:1.1.1.RELEASE:compile
[INFO] | | \-
org.springframework.security:spring-securityrsa:jar:1.0.1.RELEASE:compile
[INFO] | | \- org.bouncycastle:bcpkixjdk15on:jar:1.47:compile
[INFO] | | \- org.bouncycastle:bcprovjdk15on:jar:1.47:compile
[INFO] | +-
org.springframework.cloud:spring-cloud-configclient:jar:1.1.2.RELEASE:compile
[INFO] | | \-
org.springframework.boot:spring-boot-autoconfigure:jar:1.3.7.RELEASE:compile
[INFO] | \- com.fasterxml.jackson.core:jacksondatabind:jar:2.6.7:compile
[INFO] | \- com.fasterxml.jackson.core:jacksoncore:jar:2.6.7:compile
[INFO] +-
org.springframework.cloud:spring-cloud-startereureka:jar:1.1.5.RELEASE:compil
e
[INFO] | +-
org.springframework.cloud:spring-cloud-netflixcore:jar:1.1.5.RELEASE:compile
[INFO] | | \- org.springframework.boot:springboot:jar:1.3.7.RELEASE:compile
```

3. Build, Release, Run

A single codebase is taken through a build process to produce a single artifact; then merged with configuration information external to the app. This is then delivered to cloud environments and run. Never change code at runtime! The notion of Build leads naturally to continuous integration (CI), since those systems provide a single location that assemble artifacts in a repeatable way.

Modern Java frameworks can produce uberjars, or the more traditional WAR file, as a single CI-friendly artifact. The Release phase merges externalized configuration (see Configuration below) with your single app artifact and dependencies like the JDK, OS, and Tomcat. The goal is to produce a release that can be executed, versioned, and rolled back. The cloud platform takes the release and handles the Run phase in a strictly separated manner.

4. Configuration

This factor is about externalizing the type of configuration that varies between deployment environments (dev, staging, prod). Configuration can be everywhere: littered among an app's code, in property sources like YAML, Java properties, environment variables (env vars), CLI args, system properties, JNDI, etc. There are various solutions — refactor your code to look for environment variables.

For simpler systems, a straightforward solution is to leverage Java's System.getenv() to retrieve one or more settings from the environment, or a Map of all keys and values present. Figure 3 is an example of this type of code.

Figure 3: A portion of POM.xml showing application dependencies

```
private String userName = System.getenv("BACKINGSERVICE_UID");
private String password = System.getenv("BACKINGSERVICE_PASSWORD");
```

For more complex systems, Spring Cloud and Spring Boot are popular choices and provide powerful capabilities for source control and externalization of configuration data.

5. Logs

Logs should be treated as event streams: a time-ordered sequence of events emitted from an application. Since you can't log to a file in a cloud, you log to stdout/stderr and let the cloud provider or related tools handle it. For example, Cloud Foundry's loggregator will turn logs into streams so they can be aggregated and managed centrally. stdout/stderr logging is simple in Java:

```
Logger log = Logger.getLogger(MyClass.class.getName());
log.setLevel(Level.ALL);
ConsoleHandler handler = new ConsoleHandler();
handler.setFormatter(new SimpleFormatter());
log.addHandler(handler);
handler.setLevel(Level.ALL);
log.fine("This is fine.");
```

6. Disposability

If you have processes that take a while to start up or shut down, they should be separated into a backing service and optimized to accelerate performance. A cloud process is disposable — it can be destroyed and created at any time. Designing for this helps to ensure good uptime and allows you to get the benefit of features like auto-scaling.

7. Backing Services

A backing service is something external your app depends on, like a database or messaging service. The app should declare that it needs a backing service via an external config, like YAML or even a source-controlled config server. A cloud platform handles binding your app to the service, ideally attaching and reattaching without restarting your app. This loose coupling has many advantages, like allowing you to use the circuit breaker pattern to gracefully handle an outage scenario.

8. Environmental Parity

Shared development and QA sandboxes have different scale and reliability profiles from production, but you can't make snowflake environments! Cloud platforms keep multiple app environments consistent and eliminate the pain of debugging environment discrepancies.

9. Administrative Processes

These are things like timer jobs, one-off scripts, and other things you might have done using a programming shell. Backing Services and other capabilities from cloud platforms can help run these, and while Java doesn't (currently) ship with a shell like Python or Ruby, the ecosystem has lots of options to make it easy to run one-off tasks or make a shell interface.

10. Port Binding

In the non-cloud world, it's typical to see several apps running in the same container, separating each app by port number and then using DNS to provide a friendly name to access. In the cloud you avoid this micromanagement — the cloud provider will manage port assignment along with routing, scaling, etc.

While it is possible to rely upon external mechanisms to provide traffic to your app, these mechanisms vary among containers, machines, and platforms. Port binding provides you full control over how your application receives and responds to requests made of it, regardless of where it is deployed.

11. Process

The original 12-factor definition here says that apps must be stateless. But some state needs to be somewhere, of course. Along these lines, this factor advocates moving any long-running state into an external, logical backing service implemented by a cache or data store.

12. Concurrency

Cloud platforms are built to scale horizontally. There are design considerations here — your app should be disposable, stateless, and use share-nothing processes. Working with the platform's process management model is important for leveraging features like auto-scale, blue-green deployment, and more.