Document:

Solution Maturity

Part:

Factors of good software

Dated:

March 2022

Company contact:

Peter Giles

Peter.giles@downerdefence.com.au

Contents

1 What is Downer’s Managed Solution Environment? 1

1.1 Core Features 1

2 Solution Compatibility Overview 2

3 First Pass Assessment Quality Factors 3

3.1 Codebase 4

3.2 Dependencies 5

3.3 Config 6

3.4 Backing services 7

3.5 Build, release, run 8

3.6 Processes 9

3.7 Port binding 10

3.8 Concurrency 11

3.9 Disposability 12

3.10 Dev/prod parity 13

3.11 Logs 15

3.12 Admin processes 16

4 Bonus Factors 17

4.1 Observable 17

4.2 Schedulable 17

4.3 Upgradable 17

4.4 Least Privilege 17

4.5 Auditable 17

4.6 Securable (Identity, Network, Scope, Certificates) 17

4.7 Measurable 18

5 Examples of good candidates for D2MSE 19

6 Examples of poor candidates for D2MSE 20

7 Things for further discussion 21

7.1 Chaos engineering – has anyone thought about what could go wrong 21

7.2 Signs of a Bad Business Partner 22

7.3 Poor Communication 22

7.4 Unaligned Mission and Vision 22

7.5 Inconsistencies 22

7.6 Vague Answers 22

7.7 Inflexibility 22

7.8 Lack of Responsibility 22

7.9 Trash-Talking Others 22

7.10 Not Respecting Your Time 23

7.11 Lack of Enthusiasm 23

7.12 No Focus on Mutual Success 23

7.13 A Focus on Grievances 23

7.14 Lack of Clarity on Deliverables 23

7.15 Rushing Things 23

7.16 Increased Focus on Competitors 23

7.17 Lack of References 23

8 Glossary 29

8.1 Definitions 29

Figures

[Figure 1 One codebase many deploys 4](https://downerdefence.sharepoint.com/teams/PlatformDevelopment/Shared%20Documents/Hosted%20solutions/Solution%20Maturity%20Assessment.docx#_Toc98226369)

[Figure 2 A production deployment attached to four backing services. 7](#_Toc98226370)

[Figure 3 separate build and run stages 8](https://downerdefence.sharepoint.com/teams/PlatformDevelopment/Shared%20Documents/Hosted%20solutions/Solution%20Maturity%20Assessment.docx#_Toc98226371)

[Figure 4 Scale out via processes 11](https://downerdefence.sharepoint.com/teams/PlatformDevelopment/Shared%20Documents/Hosted%20solutions/Solution%20Maturity%20Assessment.docx#_Toc98226372)

Tables

[Table 1: A table 32](#_Toc98226366)

# What is Downer’s Managed Solution Environment?

Downer’s managed solutions provide hosted Software as a Service (SaaS), application services hosting platform environment to allow consumer business solutions for systems that process information up to and including the PROTECTED classification operated from Microsoft’s Azure platform. Unlike the Infrastructure as a Service (IaaS) offering provided by cloud service providers like Microsoft Azure, Amazon Web Services (AWS), and Vault Systems, the D2MSE takes care of not only the infrastructure hosting but also support, maintenance and consumer connectivity for the services and solutions it offers.

## Core Features

* **Safe.** The default deployment model ensures all data is replicated across multiple geographically

diverse datacentres and their deployment approach ensure the service can be reconstituted to an exact state if disaster strikes.

* **Sovereign.** The service and all its data are hosted and run within the Australian data centres by

an experienced and security cleared team.

* **Secure.** It is designed and built based on the Australian Government’s Information

Security Manual (ISM) and uses modern automation practices to manage deployment and

operations.

* **Configuration as Code** - All infrastructure is scripted as desired state configuration code and managed through source control.
* **~~PaaS First~~** ~~– Where possible solutions are architected to prioritise the use of PaaS services.~~
* **Repair by Replacement** – Patching is not done in the ‘run’ environment, Solution Infrastructure is articulated as desired state configuration code. Environments are mutated by applying the desired state to the ‘run’ environment via agents and pipelines after change approval criteria are met.
* **Privileged User Free / Hands off Operations** – Operations staff have read-only access to the environment and monitor using aggregated logging dashboards. Admin (one-off) tasks are scripted and delivered via a change approval process. Solution Vendors and Users do not have access to hosted Solution infrastructure.

Compatible solutions would answer ‘No’ to the following Questions

* Does the solution require Cloud Infrastructure that is NOT available in Australian Datacentres?
* Does the solution require access that breaches the ISM guidelines?
* Does the solution require patching in the ‘run’ environment?
* Does the solution require interactive Admin access in the ‘run’ environment?

# Solution Compatibility Overview

Broadly speaking, these factors contribute to four critical areas of modern software engineering:

* **Continuous Integration/Deployment** – automatically validate & deploy new code, build a single package for each path-to-production chain, etc.
* **Monitoring/Observability** – primarily through the focus on logs.
* **Scalability** – enable applications to support dynamic load increases through resource addition (and vice-versa), and by externalizing the system state & configuration.
* **Technical Debt Management** – through improved build/dependency debt management, and the emphasis on loosely-coupled architectures.

Solution Vendor Teams that demonstrate a good understanding of these concepts and allocate time and resources to addressing them in their product development are good candidates for strategic partnership.

# First Pass Assessment Quality Factors

Discussing these factors can reveal how prepared a solution is for multiple customer production level operations. This shouldn’t be considered a tick and flick pass or fail list. Solutions that achieve most of the Core Factors are more likely to be compatible with Cloud provisioned Infrastructure.

|  |  |  |
| --- | --- | --- |
| Core Factors | | |
| **#** | **Factor** | **Description** |
| 1 | Codebase | There should be exactly one codebase for a deployed service with the codebase being used for many deployments. |
| 2 | Dependencies | All dependencies should be declared, with no implicit reliance on system tools or libraries. |
| 3 | Config | The configuration that varies between deployments should be stored in the environment. |
| 4 | Backing services | All backing services are treated as attached resources and attached and detached by the execution environment. |
| 5 | Build, release, run | The delivery pipeline should strictly consist of build, release, run. |
| 6 | Processes | Applications should be deployed as one or more stateless processes with persisted data stored on a backing service. |
| 7 | Port binding | Self-contained services should make themselves available to other services by specified ports. |
| 8 | Concurrency | Concurrency is advocated by scaling individual processes. |
| 9 | Disposability | Fast start-up and shutdown are advocated for a more robust and resilient system. |
| 10 | Dev/Prod parity | All environments should be as similar as possible. |
| 11 | Logs | Applications should produce logs as event streams and leave the execution environment to aggregate. |
| 12 | Admin Processes | Any needed admin tasks should be kept in source control and packaged with the application. |
| Bonus Factors for Consideration | | |
| 13 | Observable | Apps should provide visibility about current health and metrics |
| 14 | Schedulable | Applications should provide guidance on expected resource constraints. i.e., Kubernetes lets you configure request limits for the containers |
| 15 | Upgradable | Apps must upgrade data formats from previous generations |
| 16 | Least privilege | Containers and Services should be running with the least privilege. Every permission you allow should be thought of as a potential attack |
| 17 | Auditable | Know what, when, who, and where for all critical operations |
| 18 | Securable | Identity, Network, Scope, Certificates - Protect your app and resources from the outsiders |
| 19 | Measurable | Application usage should be measurable for quota or chargebacks |

## Codebase

One codebase tracked in revision control, many deploys

The application is always tracked in a version control system, such as Git, Mercurial, or Subversion. A copy of the revision tracking database is known as a code repository, often shortened to code repo or just repo.

A codebase is any single repo (in a centralized revision control system like Subversion), or any set of repos who share a root commit (in a decentralized revision control system like Git).

Shape

Description automatically generated with medium confidenceOne codebase must map to many deploys

Figure 1 One codebase many deploys

There is always a one-to-one correlation between the codebase and the app:

* If there are multiple codebases, it’s not an app – it’s a distributed system. Each component in a distributed system is an app, and each can individually comply with these quality factors.
* Multiple apps sharing the same code is a violation of these quality factors. The solution here is to organise shared code into libraries which can be included through the dependency manager.

There is only one codebase per app, but there will be many deployments of the app. A deployment is a running instance of the app. This is typically a production site, and one or more staging sites. Additionally, every developer has a copy of the app running in their local development environment, each of which also qualifies as a deploy.

The codebase is the same across all deploys, although different versions may be active in each deployment. For example, a developer has some commits not yet deployed to staging; staging has some commits not yet deployed to production. But they all share the same codebase, thus making them identifiable as different deploys of the same app.

Compatible solutions would answer ‘Yes’ to the following Questions

* Is each solution component’s code maintained as a single codebase that can be deployed multiple times?
* Are customer-specific one-off features maintained in their codebase and deployed as modules/plugins through configuration?

## Dependencies

Explicitly declare and isolate dependencies

Most programming languages offer a packaging system for distributing support libraries, such as CPAN for Perl or Rubygems for Ruby. Libraries installed through a packaging system can be installed system-wide (known as “site-packages”) or scoped into the directory containing the app (known as “bundling”).

A good app never relies on the implicit existence of system-wide packages. It declares all dependencies, completely and exactly, via a dependency declaration manifest. Furthermore, it uses a dependency isolation tool during execution to ensure that no implicit dependencies “leak in” from the surrounding system. The full and explicit dependency specification is applied uniformly to both production and development.

For example, Bundler for Ruby offers the Gemfile manifest format for dependency declaration and bundle exec for dependency isolation. In Python there are two separate tools for these steps – Pip is used for declaration and Virtualenv for isolation. Even C has Autoconf for dependency declaration, and static linking can provide dependency isolation. No matter what the toolchain, dependency declaration and isolation must always be used together – only one or the other is not sufficient to satisfy these-factor.

One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app. The new developer can check out the app’s codebase onto their development machine, requiring only the language runtime and dependency manager installed as prerequisites. They will be able to set up everything needed to run the app’s code with a deterministic build command. For example, the build command for Ruby/Bundler is bundle install, while for Clojure/Leiningen it is lein dependencies.

Applications also do not rely on the implicit existence of any system tools. Examples include shelling out to ImageMagick or curl. While these tools may exist on many or even most systems, there is no guarantee that they will exist on all systems where the app may run in the future, or whether the version found on a future system will be compatible with the app. If the app needs to shell out to a system tool, that tool should be bundled into the app.

Compatible solutions would answer ‘Yes’ to the following Questions

* Does the application explicitly declare its dependencies?
* Does the application have documentation or scripts to ensure the environment has all external prerequisites available?
* Are prerequisite version dependencies tested at start-up?

## Config

Store configuration in the environment

An app’s configuration encompasses everything likely to vary between deploys (staging, production, developer environments, etc). This includes:

* Resource handles to the database, Memcached, and other backing services
* Credentials to external services such as Amazon S3 or Twitter
* Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store configuration as constants in the code. This is a violation of this principle, which requires strict separation of config from code. Config varies substantially across deploys, code does not.

A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.

Note that this definition of “config” does not include internal application config, such as config/routes.rb in Rails, or how code modules are connected in Spring. This type of config does not vary between deploys, and so is best done in the code.

Another approach to config is the use of config files that are not checked into revision control, such as config/database.yml in Rails. This is a huge improvement over using constants that are checked into the code repo, but still has weaknesses: it’s easy to mistakenly check-in a config file to the repo; there is a tendency for config files to be scattered about in different places and different formats, making it hard to see and manage all the config in one place. Further, these formats tend to be language- or framework-specific.

The app stores config in environment variables (often shortened to env vars or env). Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

Another aspect of config management is grouping. Sometimes apps batch config into named groups (often called “environments”) named after specific deploys, such as the development, test, and production environments in Rails. This method does not scale cleanly: as more deploys of the app are created, new environment names are necessary, such as Staging or QA. As the project grows further, developers may add their special environments like joes-staging, resulting in a combinatorial explosion of config which makes managing deploys of the app very brittle.

Env vars are granular controls, each fully orthogonal to other env vars. They are never grouped as “environments”, but instead are independently managed for each deployment. This is a model that scales up smoothly as the app naturally expands into more deploys over its lifetime.

Compatible solutions would answer ‘Yes’ to the following Questions

* Does the application’s configuration get all its configuration from the run environment?
* Is the application’s configuration completely documented?

## Backing services

Treat backing services as attached resources

A backing service is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd), SMTP services for outbound email (such as Postfix), and caching systems (such as Memcached).

Backing services like the database are traditionally managed by the same systems administrators who deploy the app’s runtime. In addition to these locally managed services, the app may also have services provided and managed by third parties. Examples include SMTP services (such as Postmark), metrics-gathering services (such as New Relic or Loggly), binary asset services (such as Amazon S3), and even API-accessible consumer services (such as Twitter, Google Maps, or Last.fm).

The code for an app makes no distinction between local and third-party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config. Deployment of the app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app’s code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change.

Each distinct backing service is a resource. For example, a MySQL database is a resource; two MySQL databases (used for sharding at the application layer) qualify as two distinct resources. The app treats these databases as attached resources, which indicates their loose coupling to the deployment they are attached to.

Shape

Description automatically generated with medium confidence

Figure 2 A production deployment attached to four backing services.

Resources can be attached to and detached from deploys at will. For example, if the app’s database is misbehaving due to a hardware issue, the app’s administrator might spin up a new database server restored from a recent backup. The current production database could be detached, and the new database attached – all without any code changes.

Compatible solutions would answer ‘Yes’ to the following Questions

* Does the application treat backing services as attached resources?
* Does the app behave appropriately if a backing service becomes detached?
* Are the admin requirements for 3rd party backing services documented and bundled with the application documentation?

## Build, release, run

Strictly separate build and run stages

A codebase is transformed into a (non-development) deploy through three stages:

* The build stage is a transform that converts a code repository into an executable bundle known as a build. Using a version of the code at a commit specified by the deployment process, the build stage fetches the vendor’s dependencies and compiles binaries and assets.
* The release stage takes the build produced by the build stage and combines it with the deployment’s current config. The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.
* The run stage (also known as “runtime”) runs the app in the execution environment, by launching some set of the app’s processes against a selected release.

Code becomes a build, which is combined with configuration to create a release.

The app uses strict separation between the build, release, and run stages. For example, it is impossible to make changes to the code at runtime, since there is no way to propagate those changes back to the build stage.

Figure 3 separate build and run stages

Deployment tools typically offer release management tools, most notably the ability to roll back to a previous release. For example, the Capistrano deployment tool stores releases in a subdirectory named releases, where the current release is a symlink to the current release directory. Its rollback command makes it easy to quickly roll back to a previous release.

Shape

Description automatically generated with medium confidenceEvery release should always have a unique release ID, such as a timestamp of the release (such as 2011-04-06-20:32:17) or an incrementing number (such as v100). Releases are an ‘append-only’ ledger, and a release cannot be mutated once it is created. Any change must create a new release.

Builds are initiated by the app’s developers whenever new code is deployed. Runtime execution, by contrast, can happen automatically in cases such as a server reboot, or a crashed process being restarted by the process manager. Therefore, the run stage should be kept to as few moving parts as possible, since problems that prevent an app from running can cause it to break in the middle of the night when no developers are on hand. The build stage can be more complex since errors are always in the foreground for a developer who is driving the deployment.

Compatible solutions would answer ‘Yes’ to the following Questions

* Does the application have a unique release ID (semantic versioning <https://semver.org/>)?
* Application releases can’t be mutated once it’s created.
* Is the rollback strategy documented and trivial to implement?

## Processes

Execute the app as one or more stateless processes

The app is executed in the execution environment as one or more processes.

In the simplest case, the code is a stand-alone script, the execution environment is a developer’s local laptop with an installed language runtime, and the process is launched via the command line (for example, python my\_script.py). On the other end of the spectrum, a production deployment of a sophisticated app may use many process types, instantiated into zero or more running processes.

Application run processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.

The memory space or filesystem of the process can be used as a brief, single-transaction cache. For example, downloading a large file, operating on it, and storing the results of the operation in the database. An application never assumes that anything cached in memory or on disk will be available on a future request or job – with many processes of each type running, chances are high that a future request will be served by a different process. Even when running only one process, a restart (triggered by code deploy, config change, or the execution environment relocating the process to a different physical location) will usually wipe out all local application state (e.g., memory and filesystem).

Asset packagers like django-asset packager use the filesystem as a cache for compiled assets. Applications do this compiling during the build stage. Asset packagers such as Jammit and the Rails asset pipeline can be configured to package assets during the build stage.

Some web systems rely on “sticky sessions” – that is, caching user session data in the memory of the app’s process and expecting future requests from the same visitor to be routed to the same process. Sticky sessions are a violation of a supportable application and should never be used or relied upon. Session state data is a good candidate for a datastore that offers time expiration, such as Memcached or Redis.

Compatible solutions would answer ‘Yes’ to the following Questions

* Does the application act as a stateless process?
* Does the application avoid the use of on in memory and machine local files caches to restore process state?
* Does the application avoid the use of sticky sessions for client session state?

## Port binding

Export services via port binding

Web apps are sometimes executed inside a webserver container. For example, PHP apps might run as a module inside Apache HTTPD, or Java apps might run inside Tomcat.

The application is completely self-contained and does not rely on the runtime injection of a webserver into the execution environment to create a web-facing service. The web app exports HTTP as a service by binding to a port and listening to requests coming in on that port.

In a local development environment, the developer visits a service URL like http://localhost:5000/ to access the service exported by their app. In deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes.

This is typically implemented by using dependency declaration to add a webserver library to the app, such as Tornado for Python, Thin for Ruby, or Jetty for Java and other JVM-based languages. This happens entirely in userspace, that is, within the app’s code. The contract with the execution environment is binding to a port to serve requests.

HTTP is not the only service that can be exported by port binding. Nearly any kind of server software can be run via a process binding to a port and awaiting incoming requests. Examples include ejabberd (speaking XMPP), and Redis (speaking the Redis protocol).

Note also that the port-binding approach means that one app can become the backing service for another app, by providing the URL to the backing app as a resource handle in the config for the consuming app.

Compatible solutions would answer ‘Yes’ to the following Questions

* Does the application export its services via port binding?

## Concurrency

Scale-out via the process model

Any computer program, once executed, is represented by one or more processes. Web apps have taken a variety of process-execution forms. For example, PHP processes run as child processes of Apache, started on-demand as needed by request volume. Java processes take the opposite approach, with the JVM providing one massive uber-process that reserves a large block of system resources (CPU and memory) on start-up, with concurrency managed internally via threads. In both cases, the running process(es) are only minimally visible to the developers of the app.

Scale is expressed as running processes, workload diversity is expressed as process types.

Graphical user interface, text, application

Description automatically generatedApplication processes are a first-class citizen. Processes in the application take strong cues from the Unix process model for running service daemons. Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a process type. For example, HTTP requests may be handled by a web process, and long-running background tasks handled by a worker process.

Figure 4 Scale out via processes

This does not exclude individual processes from handling their internal multiplexing, via threads inside the runtime VM, or the async/event model found in tools such as EventMachine, Twisted, or Node.js. But an individual VM can only grow so large (vertical scale), so the application must also be able to span multiple processes running on multiple physical machines.

The process model truly shines when it comes time to scale out. The share-nothing, horizontally partitionable nature of well-designed app processes means that adding more concurrency is a simple and reliable operation. The array of process types and the number of processes of each type is known as process formation.

Application processes should never daemonize or write PID files. Instead, rely on the operating system’s process manager (such as systemd, a distributed process manager on a cloud platform, or a tool like Foreman in development) to manage output streams, respond to crashed processes, and handle user-initiated restarts and shutdowns.

Compatible solutions would answer ‘Yes’ to the following Questions

* Do the application’s workloads take advantage of processes to enable scaling?

## Disposability

Maximize robustness with fast start-up and graceful shutdown

The application’s processes are disposable, meaning they can be started or stopped at a moment’s notice. This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.

Processes should strive to minimize start-up time. Ideally, a process takes a few seconds from the time the launch command is executed until the process is up and ready to receive requests or jobs. Short start-up time provides more agility for the release process and scaling up; and it aids robustness because the process manager can more easily move processes to new physical machines when warranted.

Processes shut down gracefully when they receive a SIGTERM signal from the process manager. For a web process, graceful shutdown is achieved by ceasing to listen on the service port (thereby refusing any new requests), allowing any current requests to finish, and then exiting. Implicit in this model is that HTTP requests are short (no more than a few seconds), or in the case of long polling, the client should seamlessly attempt to reconnect when the connection is lost.

For a worker process, a graceful shutdown is achieved by returning the current job to the work queue. For example, on RabbitMQ the worker can send a NACK; on Beanstalkd, the job is returned to the queue automatically whenever a worker disconnects. Lock-based systems such as Delayed Job need to be sure to release their lock on the job record. Implicit in this model is that all jobs are re-entrant, which typically is achieved by wrapping the results in a transaction or making the operation idempotent.

Processes should also be robust against sudden death, in the case of a failure in the underlying hardware. While this is a much less common occurrence than a graceful shutdown with SIGTERM, it can still happen. A recommended approach is the use of a robust queueing backend, such as Beanstalkd, that returns jobs to the queue when clients disconnect or time out. Either way, the app is architected to handle unexpected, non-graceful terminations. The crash-only design takes this concept to its logical conclusion.

Crash-only software refers to computer programs that handle failures by simply restarting, without attempting any sophisticated recovery. Correctly written components of crash-only software can micro-reboot to a known-good state without the help of a user. Since failure-handling and normal start-up use the same methods, this can increase the chance that bugs in failure-handling code will be noticed, except when there are leftover artifacts, such as data corruption from a severe failure, that don't occur during normal start-up.

Crash-only software also has benefits for end-users. All too often, applications do not save their data and settings while running, only at the end of their use. For example, word processors usually save settings when they are closed. A crash-only application is designed to save all changed user settings soon after they are changed, so that the persistent state matches that of the running machine. No matter how an application terminates (be it a clean close or the sudden failure of a laptop battery), the state will persist.

Compatible solutions would answer ‘Yes’ to the following Questions

* Can the application dispose of its processes quickly and reliably?
* Can the application start and service requests quickly?
* Can the application be shut down gracefully?
* When the application unexpectedly terminates will the application continue processing where it left off?
* Can the application micro-reboot to a known-good state without the help of a user.

## Dev/prod parity

Keep development, staging, and production as similar as possible

Historically, there have been substantial gaps between development (a developer making live edits to a local deployment of the app) and production (a running deployment of the app accessed by end-users). These gaps manifest in three areas:

* The time gap: A developer may work on code that takes days, weeks, or even months to go into production.
* The personnel gap: Developers write code, ops engineers deploy it.
* The tools gap: Developers may be using a stack like Nginx, SQLite, and OS X, while the production deployment uses Apache, MySQL, and Linux.

The application is designed for continuous deployment by keeping the gap between development and production small. Looking at the three gaps described above:

* Make the time gap small: a developer may write code and have it deployed hours or even just minutes later.
* Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behaviour in production.
* Make the tools gap small: keep development and production as similar as possible.

Table 1 Benefits of continuous deployment

|  |  |  |
| --- | --- | --- |
|  | Traditional app | Quality-factor app |
| Time between deploys | Weeks | Hours |
| Code authors vs code deployers | Different people | Same people |
| Dev vs production environments | Divergent | As similar as possible |

Backing services, such as the app’s database, queueing system, or cache, is one area where dev/prod parity is important. Many languages offer libraries that simplify access to the backing service, including adapters to different types of services.

Table 2 Backing Service Examples

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Language | Library | Adapters |
| Database | Ruby/Rails | ActiveRecord | MySQL, PostgreSQL, SQLite |
| Queue | Python/Django | Celery | RabbitMQ, Beanstalkd, Redis |
| Cache | Ruby/Rails | ActiveSupport::Cache | Memory, filesystem, Memcached |

Developers sometimes find great appeal in using a lightweight backing service in their local environments, while a more serious and robust backing service will be used in production. For example, using SQLite locally and PostgreSQL in production; or local process memory for caching in development and Memcached in production.

Developers should resist the urge to use different backing services between development and production, even when adapters theoretically abstract away any differences in backing services. Differences between backing services mean that tiny incompatibilities crop up, causing code that worked and passed tests in development or staging to fail in production. These types of errors create friction that disincentivizes continuous deployment. The cost of this friction and the subsequent dampening of continuous deployment is extremely high when considered in aggregate over the lifetime of an application.

Lightweight local services are less compelling than they once were. Modern backing services such as Memcached, PostgreSQL, and RabbitMQ are not difficult to install and run thanks to modern packaging systems, such as Homebrew and apt-get. Alternatively, declarative provisioning tools such as Chef and Puppet combined with lightweight virtual environments such as Docker and Vagrant allow developers to run local environments which closely approximate production environments. The cost of installing and using these systems is low compared to the benefit of dev/prod parity and continuous deployment.

Adapters to different backing services are still useful because they make porting to new backing services relatively painless. But all deploys of the app (developer environments, staging, production) should be using the same type and version of each of the backing services.

Compatible solutions would answer ‘Yes’ to the following Questions

* Is the delta between environments as small as possible?
* Are the backing services supporting environments using the same technology?
* Are the tools used in each environment the same?

## Logs

Treat logs as event streams

Logs provide visibility into the behaviour of a running app. In server-based environments, they are commonly written to a file on disk (a “logfile”), but this is only an output format.

Logs are the stream of aggregated, time-ordered events collected from the output streams of all running processes and backing services. Logs in their raw form are typically a text format with one event per line (though backtraces from exceptions may span multiple lines). Logs have no fixed beginning or end but flow continuously as long as the app is operating.

The application never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to stdout. During local development, the developer will view this stream in the foreground of their terminal to observe the app’s behaviour.

In staging or production deploys, each process’ stream will be captured by the execution environment, collated together with all other streams from the app, and routed to one or more final destinations for viewing and long-term archival. These archival destinations are not visible to or configurable by the app and instead are completely managed by the execution environment. Open-source log routers (such as Logplex and Fluentd) are available for this purpose.

The event stream for an app can be routed to a file or watched via real-time tail in a terminal. Most significantly, the stream can be sent to a log indexing and analysis systems such as Splunk, or a general-purpose data warehousing system such as Hadoop/Hive. These systems allow for great power and flexibility for introspecting an app’s behaviour over time, including:

Finding specific events in the past.

Large-scale graphing of trends (such as requests per minute).

Active alerting according to user-defined heuristics (such as an alert when the quantity of errors per minute exceeds a certain threshold).

Compatible solutions would answer ‘Yes’ to the following Questions

* Does the application log to stdout or the run environment equivalent?

## Admin processes

Run admin/management tasks as one-off processes

The process formation is the array of processes that are used to do the app’s regular business (such as handling web requests) as it runs. Separately, developers will often wish to do one-off administrative or maintenance tasks for the app, such as:

* Running database migrations (e.g. manage.py migrate in Django, rake db:migrate in Rails).
* Running a console (also known as a REPL shell) to run arbitrary code or inspect the app’s models against the live database. Most languages provide a REPL by running the interpreter without any arguments (e.g., python or perl) or in some cases have a separate command (e.g., irb for Ruby, rails console for Rails).
* Running one-time scripts committed into the app’s repo (e.g., php scripts/fix\_bad\_records.php).

One-off admin processes should be run in an identical environment as the regular long-running processes of the app. They run against a release, using the same codebase and config as any process run against that release. Admin code must ship with application code to avoid synchronization issues.

The same dependency isolation techniques should be used on all process types. For example, if the Ruby web process uses the command bundle exec thin start, then a database migration should use bundle exec rake db:migrate. Likewise, a Python program using Virtualenv should use the bundled bin/python for running both the Tornado web server and any manage.py admin processes.

Applications should strongly favour languages that provide a REPL shell out of the box, and which make it easy to run one-off scripts. In a local deployment, developers invoke one-off admin processes by a direct shell command inside the app’s checkout directory. In a production deployment, developers can use ssh or other remote command execution mechanism provided by that deployment’s execution environment to run such a process.

Compatible solutions would answer ‘Yes’ to the following Questions

* Are admin tasks run as one-off processes?
* Are they run against a release utilising the same environmental configuration as the application?
* Do they use the same dependencies?
* Do maintenance tasks get checked into source control as scripts?

# Bonus Factors

## Observable

Apps should provide visibility about current health and metrics

Distributed systems can be a challenge to manage because multiple microservices work together to build an application. Essentially, many moving parts need to work together for a system to function. If one microservice fails, the system needs to detect it and fix it automatically.

## Schedulable

Applications should provide guidance on expected resource constraints

Let's say that management picks your team to experiment with a project on Kubernetes. Your team works hard setting up the environment, and you end up with an application that is running with exemplary response time and performance. Another team then follows your lead—creates their application and hosts in the same environment. When the second application goes live, the original application starts experiencing performance degradation. When you start to troubleshoot, the first place to look is the compute resource assigned (CPU and memory) to your containers. Your containers are likely starving for ‘compute’ resources, and that leads to the question of how you can ensure compute resources for your applications.

## Upgradable

Apps must upgrade data formats from previous generations

Security or feature patches are often needed for applications running in production, and production applications need to upgrade without service disruption. Kubernetes provides rolling updates for applications to upgrade with no service outage. With rolling updates, you can update one pod at a time without taking down the entire service.

## Least Privilege

Containers should be running with the least privilege

We must approach access pessimistically. Every permission allowed in a container is a potential attack. For example, if your container is running with root privileges, then anyone with access to your container can inject a malicious process into it. Kubernetes provides Pod Security Policies (PSP) that you can use to restrict access to your filesystem, host port, Linux capabilities, and more.

## Auditable

Know what, when, who, and where for all critical operations

Auditability is critical for any actions performed on Kubernetes clusters or at the application. For example, if your application handles credit card transactions, you need to enable auditing to keep audit trails of each transaction.

## Securable (Identity, Network, Scope, Certificates)

Protect apps and resources from the outsiders

Applications need end-to-end security when running in production:

* Authentication: Confirm identities
* Authorization: Validate what authenticated users can access
* Certificate management: Manage digital certificates, including creation, storage, and renew
* Data protection: Security measures for data in transit and at rest
* Network security and isolation: Prevent unauthorized users and processes from accessing the network
* Vulnerability Advisor: Identify any security vulnerabilities in the images
* Mutation Advisor: Identify any mutation in containers

## Measurable

Application usage should be measurable for quota or chargebacks

The platform runner must manage costs. The compute resources allocated to run the containers should be measurable, and organizations using the cluster should be accountable.

Compatible solutions would answer ‘Yes’ to the following Questions

* Does the application run in Kubernetes or a similar container orchestration platform?

# Examples of good candidates for D2MSE

A vendor who has had a stable product for some time and is used by a few customers reliably.   
  
They can create new instances of their service for new customers quickly and can recover from chaos and disaster quickly and reliably. This process is scripted for reliability and repeatability. Configuration is managed in the deployment environment.

There are clearly understood and monitored logging features that are streamed to an operations dashboard.

They have good relationships with their current customers and are now finding that the popularity of their product necessitates a bigger commitment of operational resources. They know how to instantiate more instances of their product and are keen to leverage our expertise at running environments to let them focus on providing improved features for their customers.

There is a product team with members who have been involved since the product’s inception. They are adding new features but also burning down their technical debt. They are aware of the compromises they have made on the journey to market and have spent some time formulating plans to address these issues. They understand the underlying technologies they have built their system on and monitor those technologies for security advisories and other maintenance issues. They actively patch and upgrade their services and platforms to minimise risk to their customer base.

Compatible solutions would answer ‘Yes’ to the following Questions and Observations

* Does the application they are wanting to host adhere to a majority of the first 12 factors?
* The answer to the application run in Kubernetes is a yes.
* Does the vendor who has had a stable product for some time and is used by a few customers reliably?
* Their customers are their greatest salesperson.
* Their product development team loves the product.

# Examples of poor candidates for D2MSE

The product is completely new and hasn’t been used in a production environment by any customer. The project to create the software may still be underway and features are being presented to the project sponsor at a rapid pace. The team hasn’t yet developed any DevOps capability and is manually dropping changes in an environment that hasn’t been rebuilt since the project kicked off.

The product team has one person who has a vague idea of how to get the environment running. Nothing is documented and the run environments require constant attention from the product team. Senior members of the product team are often taken off the project to service other customers and the turnaround on technical questions is not timely and getting their full attention is difficult.

Logging is done to files on the virtual machine instances host services rather than using backing services in the environment and there are no dashboards to give an overview of system health.   
  
The solution is unable to scale, and jury-rigged fixes are put in place to get it over the line. Examples of this would be a message queue that will be used hosted on the same server that hosts the solution’s web application to reduce the processing load of long-running requests.

The underlying technologies are not understood and there is no monitoring of changes in those technologies. Open-source projects that are no longer maintained are used as core components of the system.

The project isn’t even complete yet and already the technologies and languages used are several revisions behind. The team are trying to meet their project schedule deadlines so they can take a paycheck and walk away from the engagement. There is no maintenance planned or resourced and there is already churn in the team.

Compatible solutions would answer ‘Yes’ to the following Questions and Observations

* Review the previous questions and you’ll have your answer.

# Things for further discussion

## Chaos engineering – has anyone thought about what could go wrong

Is my Frequently Asked Questions (FAQ) handbook rich enough to get solutions for most of the issues?

Are my Standard Operating Procedures (SOPs) comprehensive and well tested to be able to help me when disaster strikes?

How good are my Business Continuity Planning (BCP) and Disaster Recovery Planning (DRP) when a large outage happens? What would be my specific role in the coordination effort to manage the issues and get the apps back in prod?

How good are my tools to spot and isolate impacted services and equipment?

Are my prod applications categorized for criticality so that the SOPs and the recovery team could focus on the most critical apps and services?

Are my apps tested for security vulnerabilities? How shielded are they from the vulnerabilities?

Is my app security team involved in the architecture and design stage of the applications? Did they do threat modelling to build security into the application?

How sensitive is my app security leadership to the OWASP[[1]](#footnote-2) top threats published from time to time? Do they make a conscious effort to protect the applications from the threats that are relevant in my context?

Is the Ops team trained and coached to look at chaos engineering as a positive step to protect the applications and minimize disruptions rather than as an unwanted, overwhelming testing exercise?

Is there an approach to move from simulated pre-prod testing to real chaos testing in production?  What would it take to make such a progression? Has my leadership made effort to train and coach the tech teams to build confidence in our applications and infrastructure?

Will I be made aware of a chaos engineering exercise at all? Ideally, I should be caught unawares and pushed to provide quick support.

Am I bound by the standard SLAs when a ‘new’ issue crops up because of the chaos engineering exercise? Why not – that’s part of the testing, isn’t it?

Do we conduct chaos engineering in an integrated manner to randomly pull down apps and infrastructure or do we conduct them in separate phases such as software services, servers, client machines, network devices, etc?

How is my Site reliability engineering (SRE) team organized – is it structured as an SRE team per platform or portfolio of applications or a separate SRE team for each application?  How will the SRE teams collaborate among themselves and the Dev/architecture team during and after the chaos engineering exercise?

Is the IT leadership willing to review the chaos engineering practices and tools to make it easy for me in Ops and all others involved in building and maintaining the apps?

And what about cloud-hosted applications? Is the cloud service provider an integral part of the chaos engineering exercise?

## Signs of a Bad Business Partner

## Poor Communication

One sign that a partnership may not work is a lack of communication. If communication isn’t clear or there is a delay in responses, then that is a sure sign that it’s best if the deal does not work out. Communication up front will reveal a lot about what it will be like to work with the partner or vendor in the future, so don’t assume communication will get better if it’s not great from the start.

## Unaligned Mission and Vision

Whether it is working with my team or a possible partnership, our mission/vision should always be aligned. There will be chaos and mismanagement if one of the people we work with goes a different route. If we’re not on the same path, we need to cancel the deal.

## Inconsistencies

The biggest warning sign when evaluating a prospective partner or vendor is inconsistencies. Vendors can indeed be disingenuous at times because circumstances simply change due to conditions they cannot control. The key is to look closely at how they deal with change. Do they take a personal sacrifice, or are they perfectly fine with you absorbing the cost?

## Vague Answers

When you ask a question and can see certain parts of it are avoided even after asking numerous different ways, this could be a signal that something isn’t right. You want to work with a partner who directly answers you no matter what you are asking. Otherwise, the partner might be secretive or dishonest.

## Inflexibility

In today’s world, when dealing with partnerships there must be a win-win for both parties. If you’re looking to work with a vendor or partner who is not flexible and provides little guarantee or assurance of success for both sides, then it might be a risky endeavour. If you’re talking to a vendor, you’re the customer and they should be able to accommodate all reasonable requests.

## Lack of Responsibility

No one is perfect 100 per cent of the time, and sometimes we slip up and make mistakes. However, if your partner, vendor, or anyone else on your team can’t admit they were wrong and point fingers at everyone else, it’s likely not the best fit. You need team members who hold themselves accountable so your business can grow and succeed, and this type of behaviour won’t get it there.

## Trash-Talking Others

If a prospective partner or vendor spends a lot of time trash-talking their competition or former clients, it’s a good sign that your partnership won’t work out. If this person is willing to trash-talk another business or client to someone they just met, they’re probably a difficult person to work with. Plus, if your partnership doesn’t work out, you’ll likely receive the same treatment.

## Not Respecting Your Time

A big red flag is if they do not respect perople’s time. If they are late, miss a call, or reschedule a meeting last minute, then that is a good sign that they probably aren’t a fit. Being lazy with time shows a lack of respect for the partnership, and you don’t care.

## Lack of Enthusiasm

If either you or your potential partner isn’t enthusiastic about a partnership, it’s likely a bad idea. A lack of enthusiasm can manifest itself as poor communication, cutting meetings and conversations short, or taking a long time to make decisions. A good partnership should generate excitement for both parties.

## No Focus on Mutual Success

When there is a lack of focus on mutual success, then it’s going to be a one-way situation that is not conducive to a good partnership. They should be discussing benefits and goals from both points of view or it won’t work.

## A Focus on Grievances

A prospective business partner who talks excessively about grievances with past employers, colleagues, or partners is a cause for concern. Everyone does this to some degree, but a relentless focus on the negative is a red flag, a sign that perhaps the problem is not with the target of the complaints but the complainer.

## Lack of Clarity on Deliverables

If you’re talking circles around one another in trying to define deliverables, you may not ever be able to pin this person down to laying everything out in black and white. That can work out for a partnership outside of business, but when it comes to responsibilities and money, everyone deserves to have the same understanding of deliverables.

## Rushing Things

When a prospective lead is far too eager to get started, the odds are high that it will only get worse when you have a working relationship. These are often the nightmarish micromanagers who will do everything last minute on their end and expect you to finish things “ASAP” on Friday at 5:57 p.m.

## Increased Focus on Competitors

If a meeting with a new vendor consists mostly of talk of their competitors, they’re most likely not confident in their product or business. If they’re confident in what they’re producing, they’ll focus on themselves and what they can do to make it the best partnership possible.

## Lack of References

Any time we work with a new vendor, we should conduct background checks and check out references. If the company has a poor background check or/and can’t provide references from recent or current customers or partners, then we should move on.

Table 3: Rick’s Checklist

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| About the Application: | Security Questionnaire | Vendor Responses | Security Assessor Comments | Risk Rating |
| Software details | Please provide the name of the application and its version number: |  |  |  |
|  | A brief description of what can be achieved/delivered by the business using this application. |  |  |  |
| Service Lines |  |  |  |  |
| System Owner |  |  |  |  |
| Business Owner | Please provide the email address of the business owner |  |  |  |
| Program Manager | Please provide the email address of the Program manager |  |  |  |
| Criticality | Is this a mission-critical application? Would there be any financial impact on Downer during its outage? |  |  |  |
|  |  |  |  |  |
| Data Classification | What type of data is collected, processed and stored by application?  E.g. **Sensitive data**: This project involves processing Personally Identifiable Information (PII), Sensitive PII, or other information our customers may consider sensitive.  **Remote access only:** All information processing occurs on the customers' systems, and this project accesses those systems remotely. |  |  |  |
| Encryption | D2MSE data in transit is encrypted by the D2MSe when exiting the customer instance app zone. Does the application encrypt data? |  |  |  |
| Encryption | D2MSE data at rest is encrypted. Does the application also encrypt data? |  |  |  |
| Encryption | Describe the unencrypted protocols used by the application that might carry sensitive information.   If yes, explain why these protocols cannot be replaced by encrypted variants, and what measures you have taken to ensure that these protocols cannot be used to endanger our confidential information. |  |  |  |
| Database protection | Is there a dedicated database used to store information processed by the application? If yes, what measures are taken to protect the database? |  |  |  |
| Asset Inventory | How many instances of this application would be deployed? |  |  |  |
| Identity and Access Management |  |  |  |  |
| Authentication & Authorisation | Does any part of the application employ username/password-based authentication? |  |  |  |
| Authentication & Authorisation | ???????  Onboarding/Offboarding of the staff/users? |  |  |  |
| Privileged Access | Does the application require local administrator privileges on a day-to-day basis? |  |  |  |
| Privileged Access | Does the administrator privileges are granular enough to have maker, checker and approver roles to reduce the possibility of fraudulent transactions? |  |  |  |
| Privileged Access | Can Multi-Factor Authentication be enforced on administrators of this product? |  |  |  |
| Session Management | How long is the session timeout? |  |  |  |
| Session Management | Does your application offer a "log out" button, when clicked, not only terminates the session (deletes cookies from the client) but also invalidates the entire session ID? |  |  |  |
| Accounting & Auditing | Does the application support various roles to enable segregation of duties and principle of least privilege? |  |  |  |
| Accounting & Auditing | Is the application capable to log security-relevant events, such as authentication, data access, etc.? |  |  |  |
| Multi-factor Authentication | Does the application support Downer ADFS MFA integration? |  |  |  |
| Operational Security |  |  |  |  |
| High Availability | Does the application offer high-availability architecture and is there a single point of failure in the current design? |  |  |  |
| Remote access | Does the application offer remote access capabilities for troubleshooting /diagnostics? |  |  |  |
| Email integration | Does this application require integration to send secure emails? |  |  |  |
| Patch Management | What is the process to receive and install application updates? |  |  |  |
| Interoperability | Does the application offer integration capabilities with other applications? If yes, please elaborate or provide an URL with further details. |  |  |  |
| Interoperability | Please provide the list of APIs and web services used by this application. |  |  |  |
| Mobile Interfaces | Is there a mobile version of the application? If yes, what are the security features of the mobile application? |  |  |  |
| User Access Reviews | Does the application provide a real-time dashboard or capability to obtain a report on users and access types provided in that application? |  |  |  |
| Vulnerability Management | Does the application provider have an easily discoverable way for external researchers to report security vulnerabilities? If yes, please share the URL. |  |  |  |
| Vulnerability Management | How is the D2MSE Support personnel notified of zero-day exploits and other security vulnerabilities within this application? |  |  |  |
| Business Continuity | How will the business recover when this software becomes unavailable or unusable? |  |  |  |
| Service Continuity | Does the application provide reports and alerts on Backup failure and success logs? |  |  |  |
| Service Continuity | D2MSE determine regions to be used. |  |  |  |
| Service Continuity | D2MSE determines an appropriate backup frequency unless the customer has an agreement, configurable. |  |  |  |
| Disaster Recovery | D2MSE include the frequency required to test the backups? |  |  |  |
| Continuous Monitoring | What triggers should be configured for alerts for suspicious behaviours. Baseline D2MSE logging: such as login failures, account lockouts, data deletion or exfiltration alerts, etc. |  |  |  |
| Security Incidents | List the email addresses of people we should contact about any security issues in the application: |  |  |  |
| Security Alerts | D2MSE |  |  |  |
| Incident response | D2MSE |  |  |  |
|  |  |  |  |  |
| Local controls bypass | Does this application bypass any security controls. |  |  |  |
| Legacy Operating Systems | Does this application require legacy operating systems i.e., such as Windows Server2016 or earlier versions? |  |  |  |
|  |  |  |  |  |
| Policies & Standards | Does the use of this application violate any of the following D2MSE standards |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Legal & Regulatory | Does this application adhere to the necessary legal, regulatory, and statutory requirements? |  |  |  |
| Policies | Does the application provider have policies and procedures in place to handle security breaches, data theft, and privacy? |  |  |  |
|  | Has this application supply chain been provided or validated? |  |  |  |
| Assurance & Certification | Does the application provider be able to provide its security and audit certifications like ISO27001, SOC2, etc.? |  |  |  |
| Application Security Testing | Can the application provider share the Penetration Test report (or any similar audit reports) to provide a reasonable assurance that security vulnerabilities are continuously being detected and remediated? |  |  |  |
| Additional Comments | Provide any additional information about the security of your application: |  |  |  |

# Glossary

This section is to provide further clarity of the terms used in this document

## Definitions

Customer

A consumer of software services hosted on D2MSE

Vendor

A supplier of a software solution hosted on D2MSE

Artefact

A deployable binary, image, configuration, code files or bundle thereof that is provided and semantically versioned by a vendor.

Run (environment)

Any environment subject to the processes and controls defined in the D2MSE solution design.

Semantic Versioning

Semantic versioning is a convention defined at <https://semver.org/>. it assigns special meaning to the numbering in artefact version numbers:  
  
Given a version number MAJOR.MINOR.PATCH, increment the:

* MAJOR version when you make incompatible API changes,
* MINOR version when you add functionality in a backwards-compatible manner, and
* PATCH version when you make backwards-compatible bug fixes.

Correct semantic versioning will act as an indicator for the necessary levels of effort involved in managing changes and impact to an instance of a deployed solution.

1. <https://owasp.org/www-project-top-ten/> [↑](#footnote-ref-2)