# **Configuration Driven Development (CDD)**

## **What is CDD?**

Traditionally we build applications like this:

* Lead architects design around business requirements
* Application is built and deployed
* Changes are done through additional components (SOLID Principles) or refactoring

With CDD we build applications as follows:

* Independent components are built first, starting at the atomic level
* An interface (JSON) is defined to compose the higher level [application]
* Combination of reusable components and JSON blueprint allows developers to build up and out

Essentially, CDD is a way of using modularity to build a loosely coupled set of components that are then composed together using a common interface.

### **Definitions**

**Problem Space**

A problem or set of problems to be solved with code, for which one is considering employing CDD. This could be within a broader code structure, or standalone.

**Use Case/Scenario**

An instance of the problem space which should be solved..

**Application**

A composition of configured components that is runnable by the application runner to fulfill the needs of a particular use case.

The nature of an application depends on the problem space. It could be a full-fledged “application” in the traditional sense (à la App Store), like an interactive dashboard. Or it could be a workflow, pipeline, or other loop iteration within a larger program.

**Application Runner**

Code that instantiates, parameterizes, builds, deploys and executes applications from their configuration file definitions.

**Stage**

A collection of components that exhibit categorically similar behavior. Stages can be acted on in linear sequence by the application runner, to enforce a logical flow.

Stage organization is not required for CDD; its utility depends on the use case.

**Configuration**

A file defining how components are customized via parameters and composed together into applications.

**Component**

A code module that encapsulates a particular well-defined behavior. Often fits neatly into a behavior stage.

**Parameter**

A small customization to a component’s behavior that doesn’t rise to the level of becoming its own behavioral component. A smart default is usually provided.

**Framework**

The whole system of CDD pieces: components, configurations, and how they should be used together to compose applications.

In service of solving Use Cases in a Problem Space, we use Configuration Driven Development.

A Configuration file defines multiple compositions of Applications/Solutions, from individual Components within Stages that are customized with Parameters.

These applications are executed by the Application Runner.

### **Real World Analogy**

Consider a “Build-A-Solution" company:

**Problem Space**

* Organizations need Solutions but they like to be able to customize many aspects of the solution for their wants and needs. However, most customers will not know the specifics of solution building procedure, components, proper build sources, etc.; they only want to specify form and function and have the details filled in for them.

**User Story/Use Case**

* A User or User Group’s particular wants and needs when they are in the market for a customizable Solution.

**Application**

* The completed Solution after it’s been built.

**Application Runner**

* The Solution Construction App that builds the Solution according to specifications.

**Stage**

* A step of the solution building process such as: foundation, structure, connectivity/data, calculation, deployment, etc.

**Configuration**

* The blueprints and plans detailing exactly how the solution should be built, what features/components it should have, and how those should be customized.

**Component**

* Features of the solution that can be customized or specified. For example, within the structure stage, there might be deployment options such as hybrid, public, or private solution. We can also optionally add a search component or change the default GUI layout to add plugins.

**Parameter**

* Customizable details of a single component; for example, the deployment-environment component could have options for region, number of instances, and private/public.

**Framework**

* The entire Build-A-Solution Service and its offerings.

## **When do we use CDD?**

Consider using CDD for a problem space.

1. There will be an unknown number of multiple iterations of similar use cases within a large or unbounded problem space. For example:
   1. A user-customizable widget-based dashboard
   2. An Extract, Transform, Load (ETL) pipeline handling multiple similar data sources
2. Applications can differ in certain *behaviors* which can be extracted into common components. They should resemble Lego blocks that are built into bigger forms. For example:
   1. Polling a data source every 30 minutes vs. accepting data change notifications
   2. Running a subset of available widgets
   3. Accepting input from command line vs. web form
3. These common components can be reused with small configuration parameter changes. For example:
   1. I/O file format
   2. Website URL
   3. Access credentials
   4. Data aggregation method
4. It’s desirable to allow non-developers to update and maintain the applications as use cases form and change.

## **When does CDD not apply?**

You would not want to use CDD:

1. If the number of use cases is known and small, or if there is an insignificant overlap between use cases, CDD likely will be more effort than it’s worth.
2. If the use cases do not have significant behavioral differences, you may be able to write one implementation with normal configurable parameters.
3. If there are no configurable parameter differences in the components, that may be OK. Or it may mean that the behaviors aren’t complex enough to warrant use of CDD.
4. If you don’t intend for non-developers to be able to update the applications, that is by no means a showstopper. But you would want to ask yourself whether CDD is worth it.

## **Why Should We Use CDD?**

These are some of the benefits.

1. There should be almost no repetition of code if organized efficiently.
2. New applications can be added very easily to solve new use cases.
3. There’s no need to worry too much about future-proofing, as new components and parameters can always be added on later.
4. It’s easy to allow non-developers to create and maintain their own applications without any code.

## **How Do We Do It?**

### **Configuration File**

The configuration file is the main aspect, hence namesake, of Configuration Driven Development. The specific layout of the file is up to the developer, but in general I find it nice to organize things by encapsulating all relevant parameters in a sub-configuration that can then be passed to the specific component. You should set a format and stick to it, otherwise the file will quickly become unreadable.

#### **Language Options**

* [JSON](https://www.json.org/json-en.html) - A popular choice for file format, due to its ubiquity and usefulness for configuration updates over the web
* [XML](https://www.w3.org/XML/) - I strongly advise against it, because it’s bad. Seriously. Please don’t use XML unless you have to!
* [YAML](https://yaml.org/) - My current go-to format, mostly due to its conciseness over JSON and ability to have comments
* [TOML](https://toml.io/en/) - Another modern configuration markup language that seems to be growing in popularity

#### **Example Configuration File in YAML**

Solution 1 Configuration:

process\_stages:

- extract:

method: extractMethodA

options:

extractMethodAOption1: false

extractMethodAOption2: 3.14

- transform: ~ *# etc., etc...*

- load: ~ *# da da da...*

Use Case 2:

~ *# and so on, and so on...*

### **Core Components**

The core components are the most important aspect to get right, following the Goldilocks Principle. Too specific and they end up not being reusable; too general and they become hard to maintain and configure. As with other architectural decisions, you will have to use discretion on a case-by-case basis to determine the appropriate size and scope of components.

How should these components be organized?

**Iterative Approach**

The key to this philosophy is to not overdesign the components at the start - it must be an iterative approach or else you lose many of the benefits of using CDD. Here’s an example workflow of iterating on a CDD project over time:

1. Receive a few initial use cases within the problem space.
   1. Break down the use cases into processing stages, and then further into behavioral components.
   2. Build behavioral components, utilizing configurable parameters liberally.
   3. Write unit tests for each component.
   4. Build a simple application runner that runs through processing stages and assembles behavioral components.
   5. Compose the initial applications in the configuration file.
   6. Write/perform functional tests of the whole application flows.
   7. Document stages, components and their accepted configuration parameters.
2. Receive additional use cases or changes to existing.
   1. Reuse existing behavioral components with additional configuration parameters if needed. Refactor as you go.
   2. Determine completely new behavioral components required; repeat steps 1.2 and 1.3.
   3. Make changes/additions to the configuration file to compose the applications.
   4. Repeat steps 1.6 and 1.7.
3. Repeat step 2 ad nauseum!

#### **Testing**

A note on testing. You can see in the iterative approach above, it’s mentioned several times. This is important because in order for the components to be generally useful, they need to work as advertised with any configuration, not just the initial application composition that drove development of the component.

# **Mechanics of Configuration**

Our discussion up to this point has covered configuration philosophy. This section shifts focus to the mechanics of how a user interacts with the configuration.

##### **Separate Configuration and Resulting Data**

Which language to store the configuration in is an inevitable question. You could choose to have pure data like in an INI, YAML, or XML file. Alternatively, the configuration could be stored in a higher-level language that allows for much more flexible configuration.

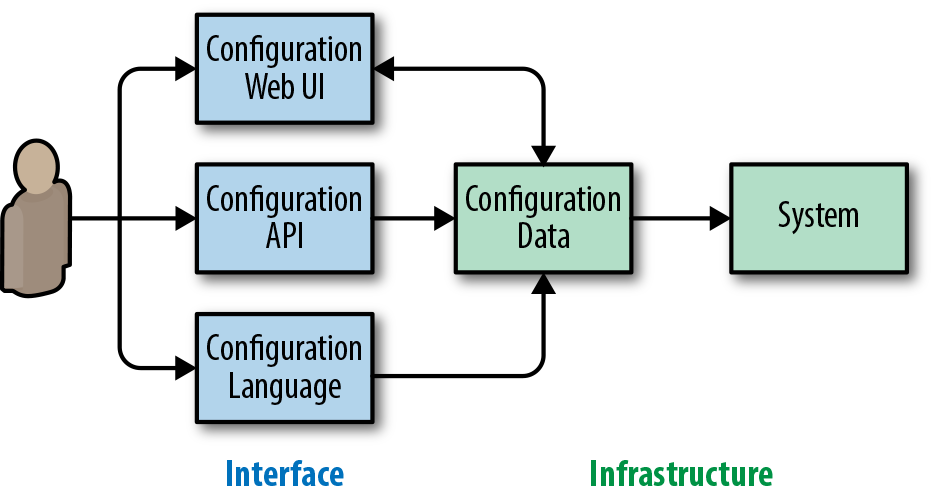
Fundamentally, all questions the user is asked boil down to static information. This may include obviously static answers to questions like “How many threads should be used?” But even “What function should be used for every request?” is just a static reference to a function.

To answer the age-old question of whether configuration is code or data, our experience has shown that having both code and data, but separating the two, is optimal. The system infrastructure should operate on plain static data, which can be in formats like [Protocol Buffers](https://developers.google.com/protocol-buffers/), [YAML](https://yaml.org/), or [JSON](https://www.json.org/). This choice does not imply that the user needs to actually interact with pure data. Users can interact with a higher-level interface that generates this data. This data format can, however, be used by APIs that allow further stacking of systems and automation.

This high-level interface can be almost anything. It can be a high-level language like Python-based Domain-Specific Language (DSL), Lua, or purpose-built languages, such as Jsonnet (which we will discuss in more detail in [Configuration Specifics](https://sre.google/workbook/configuration-specifics/)). We can think of such an interface as a compilation, similar to how we treat C++ code.[4](https://sre.google/workbook/configuration-design/#ch14fn4) The high-level interface might also be no language at all, with the configuration ingested by a web UI.

Starting with a configuration UI that’s deliberately separated from its static data representation means the system has flexibility for deployment. Various organizations may have different cultural norms or product requirements (such as using specific languages within the company or needing to externalize configuration to end users), and a system this versatile can be adapted to support diverse configuration requirements. Such a system can also effortlessly support multiple languages.[5](https://sre.google/workbook/configuration-design/#ch14fn5) See [Figure 14-2](https://sre.google/workbook/configuration-design/#configuration-flow-with-a-separate-configuration-interface-and-configuration-data-infrastructure).

This separation can be completely invisible to the user. The user’s common path may be to edit files in the configuration language while everything else happens behind the scenes. For example, once the user submits changes to the system, the newly stored configuration is automatically compiled into raw data.[6](https://sre.google/workbook/configuration-design/#ch14fn6)

**Figure 14-2. Configuration flow with a separate configuration interface and configuration data infrastructure. Note the web UI typically also displays the current configuration, making the relationship bidirectional.**

Once the static configuration data is obtained, it can also be used in data analysis. For instance, if the generated configuration data is in JSON format, it can be [loaded into PostgreSQL and analyzed with database queries](https://www.postgresql.org/docs/10/static/functions-json.html). As the infrastructure owner, you can then quickly and easily query for which configuration parameters are being used and by whom. This query is useful for identifying features you can remove or measuring the impact of a buggy option.

When consuming the final configuration data, you will find it useful to also store metadata about how the configuration was ingested. For example, if you know the data came from a configuration file in Jsonnet or you have the full path to the original before it was compiled into data, you can track down the configuration authors.

It is also acceptable for the configuration language to be static data. For example, both your infrastructure and interface might use plain JSON. However, avoid tight coupling between the data format you use as the interface and the data format you use internally. For example, you may use a data structure internally that contains the data structure consumed from configuration. The internal data structure might also contain completely implementation-specific data that never needs to be surfaced outside of the system.

##### **Importance of Tooling**

Tooling can make the difference between a chaotic nightmare and a sustainable and scalable system, but it is often overlooked when configuration systems are designed. This section discusses the key tools that should be available for an optimal configuration system.

###### Semantic validation

While most languages offer syntax validation out of the box, don’t overlook semantic validation. Even if your configuration is syntactically valid, is it likely to do useful things? Or did the user reference a nonexistent directory (due to a typo), or need a thousand times more RAM than they actually have (because units aren’t what the user expected)?

Validating that the configuration is semantically meaningful, to the maximum extent possible, can help prevent outages and decrease operational costs. For every possible misconfiguration, we should ask ourselves if we could prevent it at the moment the user commits the configuration, rather than after changes are submitted.

###### Configuration syntax

While it’s key to ensure that configuration accomplishes what the user wants, it is also important to remove mechanical obstacles. From a syntax perspective, the configuration language should offer the following:

*Syntax highlighting in editors (used within the company)*

* Often, you’ve already solved this by reusing an existing language. However, domain-specific languages may have additional “syntactic sugar” that can benefit from specialized highlighting.

*Linter*

* Use a linter to identify common inconsistencies in language use. [Pylint](https://www.pylint.org/) is one popular language example.

*Automatic syntax formatter*

* Built-in standardization minimizes relatively unimportant discussions about formatting and decreases cognitive load as contributors switch projects. Standard formatting may also allow for easier automatic editing, which is helpful in systems used broadly within a large organization. Examples of autoformatters in existing languages include clang-format[7](https://sre.google/workbook/configuration-design/#ch14fn7) and [autopep8](https://pypi.org/project/autopep8/).

These tools enable users to write and edit configuration with confidence that their syntax is correct.[8](https://sre.google/workbook/configuration-design/#ch14fn8) Incorrect indentation in whitespace-oriented configs can have potentially great consequences—some of which standard formatting can prevent.

##### **Ownership and Change Tracking**

Because configuration can potentially impact critical systems of companies and institutions, it’s important to ensure good user isolation, and to understand what changes happened in the system. As mentioned in [Postmortem Culture: Learning from Failure](https://sre.google/workbook/postmortem-culture/), an effective postmortem culture avoids blaming individuals. However, it’s helpful both during an incident and while you’re conducting a postmortem to know who changed a configuration, and to understand how the configuration change impacted the system. This holds true whether the incident is due to an accident or a malicious actor.

Each configuration snippet for the system should have a clear owner. For example, if you use configuration files, their directories might be owned by a single production group. If files in a directory can only have one owner, it’s much easier to track who makes changes.

Versioning configuration, regardless of how it is performed, allows you to go back in time to see what the configuration looked like at any given point in time. Checking configuration files into a versioning system, such as Subversion or Git, is a common practice nowadays, but this practice is equally important for configuration ingested by web UI or remote APIs. You may also wish to have tighter coupling between the configuration and the software being configured. By doing so, you can avoid inadvertently configuring features that are either not yet available or no longer supported in the software.

On a related note, it is useful (and sometimes required) to log both changes to the configuration and the resulting application to the system. The simple act of committing a new version of a configuration does not always mean that the configuration is directly applied (more on that later). When a system configuration change is suspected as the culprit during an incident response, it is useful to be able to quickly determine the full set of configuration edits that went into the change. This enables confident rollbacks, and the ability to notify parties whose configurations were impacted.

##### **Safe Configuration Change Application**

As discussed earlier, configuration is an easy way to make large changes to system functionality, but it is often not unit-tested or even easily testable. Since we want to avoid reliability incidents, we should inspect what the safe application of a configuration change means.

For a configuration change to be safe, it must have three main properties:

* The ability to be deployed gradually, avoiding an all-or-nothing change
* The ability to roll back the change if it proves dangerous
* Automatic rollback (or at a minimum, the ability to stop progress) if the change leads to loss of operator control

When deploying a new configuration, it is important to avoid a global all-at-once push. Instead, push the new configuration out gradually—doing so allows you to detect issues and abort a problematic push before causing a 100% outage. This is one reason why tools such as Kubernetes use a rolling update strategy for updating software or configuration instead of updating every pod all at once. (See [Canarying Releases](https://sre.google/workbook/canarying-releases/) for related discussions.)

The ability to roll back is important for decreasing incident duration. Rolling back the offending configuration can mitigate an outage much more quickly than attempting to patch it with a temporary fix—there is inherently lower confidence that a patch will improve things.

In order to be able to roll forward and roll back configuration, it must be hermetic. Configuration that requires external resources that can change outside of its hermetic environment can be very hard to roll back. For example, configuration stored in a version control system that references data on a network filesystem is not hermetic.

Last but not least, the system should be especially careful when handling changes that might lead to sudden loss of operator control. On desktop systems, screen resolution changes often prompt a countdown and reset if a user does not confirm changes. This is because an incorrect monitor setting might prevent the user from reverting the change. Similarly, it is common for system admins to accidentally firewall themselves out of the system that they are currently setting up.

These principles are not unique to configuration and apply to other methods of changing deployed systems, such as upgrading binaries or pushing new data sets.

Reference: <https://sre.google/workbook/configuration-specifics/>