Chapter 11

Stage 5: Consolidating Knowledge in Builds

11.1 INTRODUCTION

As analytics code is written by the team to produce work products, common patterns of data manipulation will begin to emerge. Here are some examples.

- **Repeated cleaning:** Every time a dataset is used, it must be prepared by cleaning data fields. For example, for a dataset of country names, the team may have to remember to produce one common version of UK, U.K., United Kingdom, and variants such as GBR and Britain.
- **Repeated joining:** Some datasets will have to be joined together. For example, every time a particular view of department and spend is required, the department and spend data sources must be joined together. This may have to be done while remembering particular rules about the join. For example, perhaps older decommissioned departments no longer have any spend data and so drop out of joins.
- **Customer inputs:** The customer or a nonanalytics team member will have inputs into the data through human keyed datasets. For example, a vendor master dataset contains 10 approximate duplicates that the customer has reviewed and identified in a list. The customer always wants these duplicates removed from analyses when counting the total number of vendors.
- **Consistent cleaning:** Imagine that as a team, you have agreed a convention of always using upper case names. For example, "Mahony" always becomes "MAHONY." This type of rule is sometimes needed for presentation purposes or to comply with algorithm inputs. The team must remember to do this consistently on appropriate fields only.
- **Need for functionality:** The need for certain code functionality will arise again and again. For example, if the team is testing data completeness they will need generic code that converts and checksums fields in datasets. If one team member has gone to the trouble of writing this code, the other team members should be able to benefit from this effort.

There are some common themes here. Both the Guerrilla Analytics team and the customer are growing their understanding of the data and this

understanding of the data needs to be captured in everything the team does for consistency.

When discoveries are made in the data they need to be communicated to everybody in the analytics team. High-profile work products delivered to a customer must include the rules the customer has agreed with the team. It reflects poorly on the team if their outputs are inconsistent or previous lessons learned appear to be forgotten at a later date.

The team will develop code to solve common data problems and this code should be reused as far as is possible. There is no point in every team member reinventing the wheel.

11.2 PITFALLS AND RISKS

Failing to maintain a consistent data understanding presents some risks to the analytics team's outputs and efficiency.

- **Copy and paste code:** Because every work product is built from scratch, there is a temptation to copy and paste code from previous work products so they can quickly be reused in the current work product. This is particularly the case when work product code is complex. Copy and paste code is inefficient in terms of the size of the team's code base and the necessary reviews. It also means that many bugs are spread through many work products. There is also a risk that old copy and pasted code does not capture the latest data and business rules.
- **Inconsistent work products:** With each team member taking their own approach to implementing manipulations of the data, there is a lack of consistency in approaches. For example, one team member's cleaning routine may remove £$%^ from text strings. Another team member's cleaning routine may remove £$%^ and in addition the symbol *. Even this simple example could result in two team members giving inconsistent counts of unique strings. If that difference translates into a difference in a material value then there are serious implications for the project.
- **Reduced efficiency from lack of abstraction:** Every team member potentially needs to know every dataset quirk and join to be able to create work products. There is no core collection of datasets that the team can trust and use without needing a full understanding of how to derive those datasets from scratch. In effect, there is no abstraction of data understanding.
- **Reduced efficiency from repeatedly building data from scratch:** Let us say it takes a significant number of data manipulations to get to a particular view of the data. Some of these data manipulations may be time consuming to run. Now, every time a work product is created that depends on that dataset, those time-consuming data manipulations must be rerun. This wastes the team's time and computational resources.

- **Data populations are not explainable to the customer:** Because every team member is producing every work product "from scratch," there is not one true view of the data that can be explained to the customer and signed off.

Clearly something needs to be done to address a lack of centralized team knowledge in a Guerrilla Analytics environment.

11.3 EXAMPLE: THE CUSTOMER ADDRESS PROBLEM

To understand the problem of evolving data understanding, it helps to visualize what is happening to the data. Figures 35 and 36 illustrate the situation for the example of a customer address problem.

The raw data consist of a customer details dataset and an address history dataset. This particular analytics job requires only the latest address for a customer. However, there are some inconsistencies and duplicates in the customer address dataset. These have been identified and manually reviewed by the broader team on two occasions and this knowledge has been captured in two duplicate review datasets. Two work products have also been created from the data on two separate occasions by two different analysts.

Figure 35 shows the approach taken by one analyst for the work product with UID 314. This analyst used the manual duplicate markups from 2nd February and immediately deduplicated the addresses. The latest address was then calculated and combined with customer details to produce a customer address dataset. At this point, some problems in customer names were noticed and so cleaning was applied to the customer address dataset to produce the final work product dataset.

Now, consider Figure 36 that is a later work product with UID 320, which was produced from the same data but by a different analyst. This analyst used

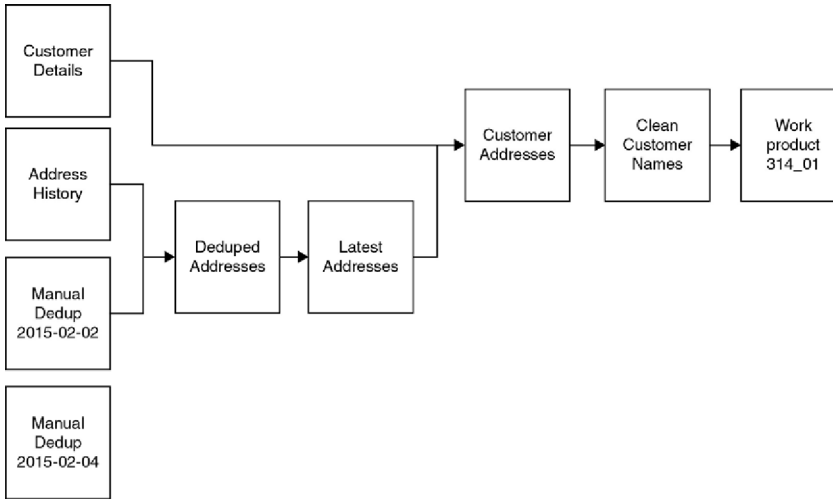

FIGURE 35 Data flow (Option 1)

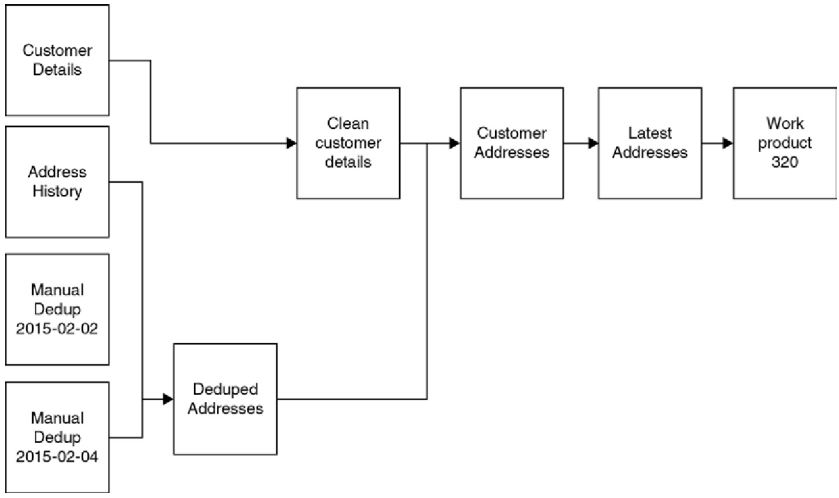

FIGURE 36 Data flow (Option 2)

a later manual deduplication file from 4th February, so the results are probably different from the previous work product 314. In addition, this second analyst has made some different design choices. They noticed issues with the customer details dataset and applied some cleaning to it. They produced a customer address dataset before calculating the latest address and then output the result into work product 320.

These two approaches to the same problem illustrate the challenge of data consistency in a very dynamic environment. The analysts have manipulated data in different orders, using slightly different inputs and perhaps different customer cleaning rules. Trying to communicate within the team the myriad ways of doing the same analyses is challenging. When larger teams are producing many work products, achieving any kind of consistency is impossible. What is needed is a way to control and version the evolving data understanding.

11.4 SOURCES OF VARIATION

If you want to control evolving data understanding, you must first identify what causes data understanding to change. Earlier in the book, we examined sources of variation in work product outputs. As a recap, these sources of variation are the following.

- **Changes in raw data:** All work products are derived from raw data provided to the team. Modify the raw data and all work products using that data are affected. You saw in the customer address example where a new deduplication mark-up changed the total number of customers identified in the population.

- **Changes to code:** A work product is created by using program code to manipulate data. That code is a reflection of both the creator's design decisions and the project's business rules that are applied to the data. If this code changes, it is very likely that the data resulting from the code will also change. You saw this in the customer address example where one analyst noticed issues with the customer details dataset and applied some custom code to fix these issues.
- **Changes to common libraries, languages, and services:** Program code will draw on libraries of common routines and services. Examples include analytics libraries, fuzzy matching libraries, and data-enrichment services. If these are changed then the data manipulation code is effectively changed too.
- **Changes to input parameters:** Some data manipulations have input parameters that change their behavior. For example, a fuzzy matching algorithm might have a threshold similarity of matched words for inclusion as a match. A machine-learning algorithm for clustering may require that a starting number of clusters be specified. Running these algorithms with different input parameters effectively changes their behavior and therefore their data outputs.
- **Nondeterministic code:** In some scenarios it is possible to write nondeterministic code. That is, code that executes differently on every execution. A typical cause of nondeterministic code is the use of random number generators. Without careful use, a random number generator in code (for example, to randomly choose a sample population) could create data that cannot ever be reproduced in the future.

To achieve consistency in the consolidated knowledge from the team and customer, you need to remove or control these sources of variation. This is achieved using something called a build. The rest of this chapter will now discuss builds. Note that this is probably the technically most challenging chapter of the book. It is worth the investment of your time as builds are by far the most significant contributor to maintaining data provenance in Guerrilla Analytics.

11.5 DEFINITION OF A BUILD

A build is centralized and version-controlled program code and data that captures the team and customer's evolving knowledge. Such knowledge about the data includes the following.

- **Rules:** The business rules and cleaning rules that must be applied to the data.
- **Convenience datasets:** The common useful views of the data that facilitate work product creation.
- **Fixes:** The agreed and signed off fixes for known data quality issues.
- **Common code:** The useful centralized code libraries that the team can use to reduce their development efforts and promote consistency.

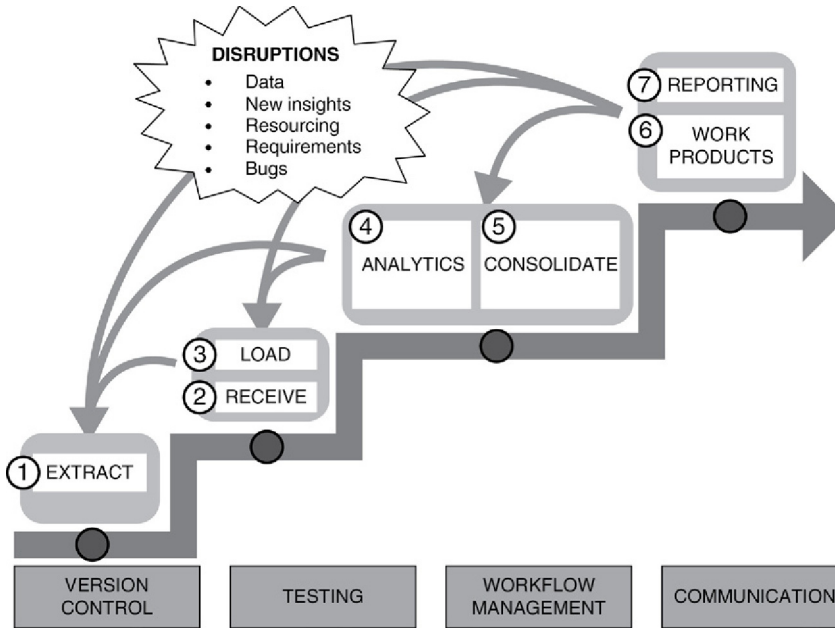

FIGURE 37 The Guerrilla Analytics workflow

- **Reference data:** The lookup datasets such as data mappings, filters, and mark-ups that feed into manipulations of other data.

Consolidation into a Build happens in Stage 5 of the Guerrilla Analytics workflow (Figure 37). We avoided discussing this stage so far as I first wanted you to understand what the full workflow looks like. Builds are best illustrated with an example, so we will now revisit the customer address problem with a build approach.

11.6 THE CUSTOMER ADDRESS EXAMPLE USING A BUILD

Consider the customer address example again. Figure 38 shows a better approach to achieving consistency by using a build. In essence, raw data is manipulated through several intermediate datasets to produce a single "official" customer address interface that is used by both work products. Here are the steps in the build.

- The first step is to create a pointer over the latest version of each type of raw dataset. This removes the problem of different analysts choosing different versions of the manual deduplication review.
- All datasets are then passed through a cleaning step. This removes the problem of inconsistent ad-hoc cleaning that could affect the outputs of the duplicate detection. While the need for cleaning may be minimal, having a

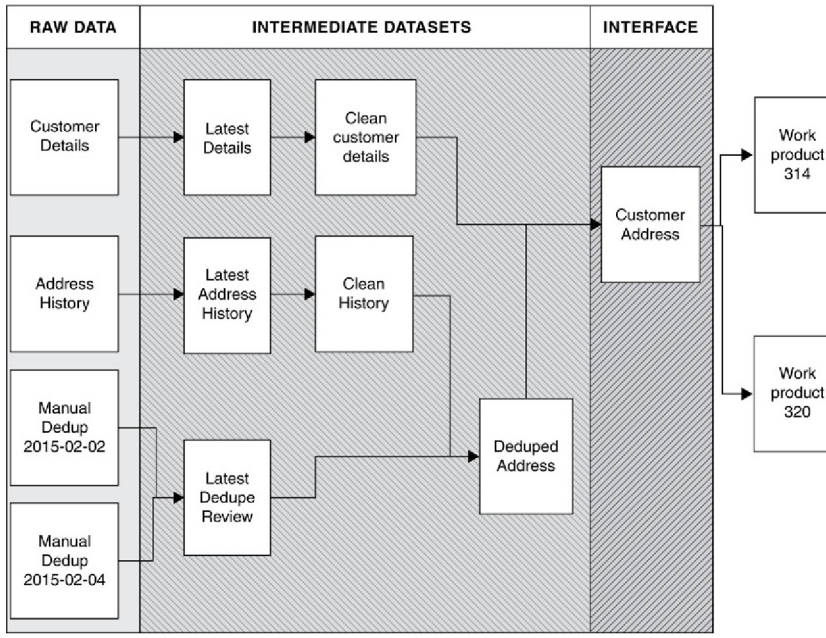

FIGURE 38 Data flow using a build

clearly defined location for cleaning in the data flow avoids ad-hoc modifications to the data throughout the data flow.

- Finally, a customer address dataset is created in an interface layer. It is this interface layer dataset that is used by all work products. This means the work products do not have to worry about the complexity of producing the customer address dataset and can instead rely on a consistent view of customer addresses.

The intermediate datasets and interface dataset in this example are a Data Build. They are produced by version-controlled code and can be completely reproduced by executing this code against the underlying raw data.

The following section discusses Data Builds and how to architect them.

11.7 DATA BUILDS

There are several points to note about how data is now flowing in the customer address Data Build.

- **The Build can be completely regenerated from raw data and Build code alone:** Build datasets are derived from the raw data using program code and that program code alone. There are no ad-hoc manual steps in the process as these steps cannot be version controlled and cannot be easily reproduced.

- **Data flows out of raw, never into raw:** No process (work product or build code) is writing back into the raw layer. Remember that a core Guerrilla Analytics principle is that raw data is never modified.
- **Some work products draw from the Build interface:** Work products that need the convenience of these Build datasets draw on them in their specific work product code.
- **Some work products still draw from raw:** There will always be work products that need to draw directly from raw datasets because these raw datasets are not yet part of any build. This is to be expected. A good build process will monitor these types of work products and include their features in the build as necessary over time.
- **Not all data is used in a build:** There are some raw datasets that are not part of any build. They might be stand alone datasets that bear no relation to anything in the build. They might be datasets that are not in frequent enough demand by the project to merit inclusion in a build.

You can now see how this Data Build promotes consistency and efficiency in the team. A work product that uses a build dataset has a head start since much of the hard work in manipulating the data into a necessary shape has been done in the build. All work products drawing on a build dataset do not need to worry about consistency as the build datasets are provided in one central and version-controlled location.

11.7.1 The Structure of a Data Build

How do you make a Data Build happen in practice? There are several components to put in place.

- **Raw data location:** There must be a Data Manipulation Environment (DME) location where raw data is stored and read by the build code. Build processes can never write into this location.
- **Intermediate and interface data location:** This is a DME location where the build code can create and delete its intermediate and interface datasets.
- **Temporary dataset location:** To avoid clutter in the DME, it is useful to have a location for temporary datasets created during the build process but not needed as a final intermediate or interface dataset.
- **Version-controlled build code:** This is a location where build program code can be stored and version controlled.

This is illustrated in Figure 39. The program code files manipulate the raw data through a data flow, turning it into several intermediate and temporary datasets until one or more interface datasets are produced. The intermediate datasets are primarily available for debugging and explaining data provenance. The rest of the team mostly uses the build datasets at the published interface. These are the datasets they use in their work product development, testing, and audit.

11.7.2 Practice Tip 45: Decouple Raw Data from Build Data

In the discussion of data flow and in the description of the Data Build structure, you saw how the starting point of any build is raw data. Using just the build program code and raw data, every dataset in the build could be generated from the raw data. You also know that a Guerrilla Analytics project is going to present the team with frequent raw data refreshes, replacements, and additions. If any of these refresh datasets contribute to the Data Build, then the build must be rerun to generate up-to-date build datasets. It would be convenient to minimize raw data references in build code so that there are minimal changes to make to the code when new data arrives.

The first stage of any build should create a pointer to the latest raw data. It is this pointer that is then referenced in the subsequent build code rather than any direct referencing of the raw data.

When new data arrives, only the pointer to the raw data needs to be changed and the rest of the build code should still execute.

Figure 39 shows pointers in the intermediate dataset layer immediately above the raw data layer.

11.7.3 Practice Tip 46: Generate Data Builds with Version-controlled Code

You have seen how new data arrives and you can decouple build code from that new data so that the build code changes required by the new data are minimized. However, your data provenance alarms should be sounding. You have changed the raw data and therefore have changed the Data Build datasets. Any work products that depend on these datasets are now broken and cannot be reproduced. You need a way to version control the creation of Data Builds so that

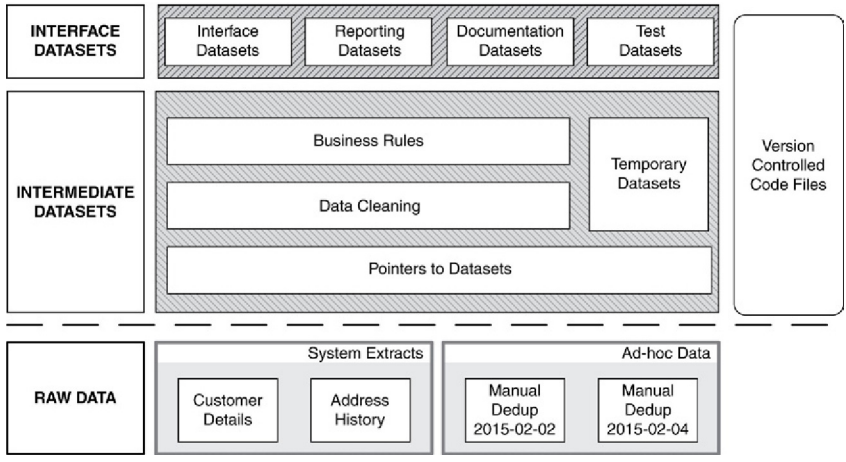

FIGURE 39 Structure of a Data Build for the customer address example

earlier versions can be recreated if necessary and the use of a particular build version can be easily identified in a work product.

Recall that every Data Build reads from raw data but never writes back to raw data. Everything flows from raw datasets through to interface datasets. This means that if you have Data Build program code and your raw data is still available, you can completely regenerate every dataset in the Data Build. With version-controlled build code, you always have a means to regenerate earlier builds as new builds evolve.

11.7.4 Practice Tip 47: Embed the Build Version in the Build Dataset Names

Over time, many versions of a Data Build will be created. Some Guerrilla Analytics projects will have over 20 versions of a build and several build projects. Work products will be derived from all of those build versions. It would save the team a lot of effort if they could easily determine the particular version of a build they are working with. A simple way to do this is to embed the Data Build version in the namespace of the build datasets.

Figure 40 shows a set of build code versions on the file system and the associated build datasets they have produced in the DME.

- Each Data Build project has its own code folder.
- You can see that each version of the build code is located in a subfolder named with the version number. In this case there are two versions of the build.
- Each build dataset has the same build name and version number embedded in its schema.

Note that this relational database example uses schemas for namespace separation. Other namespace separations are possible in non-relational DMEs.

Work product SQL code that uses this build's datasets from version 2 would look like the following.

```
Select CUSTOMER_NAME
FROM CUSTOMER_ADDRESSES_002.CUSTOMER_ADDRESS
```

Without any documentation overhead, a reader of the code can tell which build (customer addresses) and which version (version 002) were used. If necessary, the correct version of the build code to recreate the dataset can be found on the file system under a folder with the build project name and build project version number.

11.7.5 Practice Tip 48: Tear Down and Rebuild Often

For each version of the build, you now have a specific destination namespace for the generated build datasets. Builds can become time consuming to execute

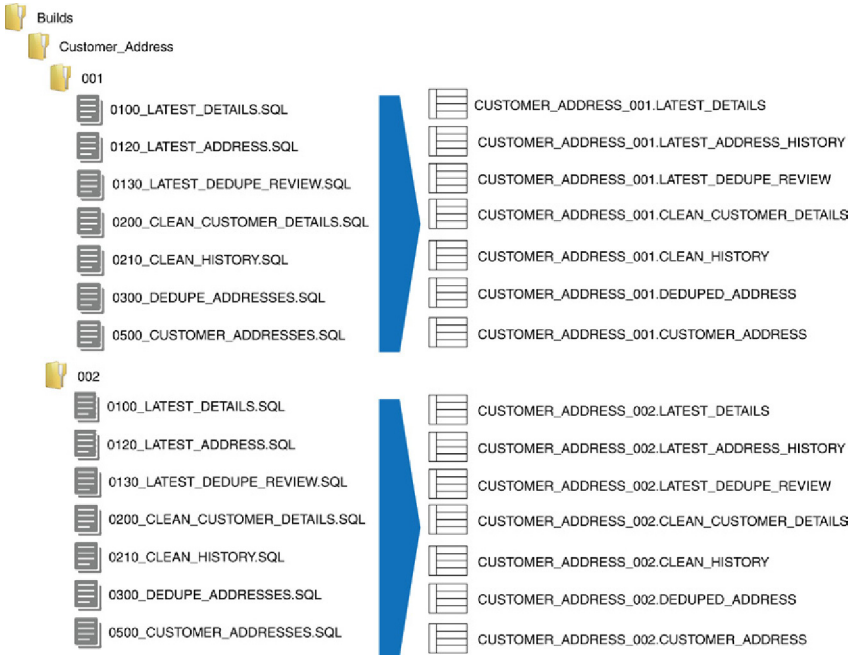

FIGURE 40 A basic version control system for a Data Build

since they are doing a lot of the "heavy lifting" for the convenience of work products. The temptation is to rerun subsets of the build's data flow and avoid regenerating all intermediate datasets. This gives rise to the legacy dataset bug, where old and out of date datasets continue to be referenced in the build code. Build code continues to execute without problems but is referencing an older incorrect dataset. Here is a simple example of how this could happen in the customer address data flow from Figure 38.

Imagine that the cleaning of the latest datasets is time consuming. The team is focusing its efforts on the deduplication code as this is more difficult to implement. The customer details dataset is refreshed so code is written to point to this latest dataset and the build is executed. This new data however, has a previously unseen quirk that requires an addition to the cleaning rules that create CLEAN_CUSTOMER_DETAILS. If this rule is not implemented, the total customers will be over counted. An analyst codes up this rule and writes the results into a new dataset called CLEAN_CUSTOMER_DETAILS_ADVANCED. However, there is now a problem. All subsequent build codes are still referencing the original CLEAN_CUSTOMER_DETAILS. This code will probably continue to execute without problems. The stale data reference may go unnoticed and the interface datasets will not reflect the advanced customer detail cleaning logic.

The simple solution to discovering and eliminating these bugs is for the build code to clear out its destination of all datasets before every execution run

of the build code. In the customer address example, a tear-down approach would have deleted all datasets including the old CLEAN_CUSTOMER_DETAILS. Subsequent code that still used references to this dataset would then have failed to execute and the stale data reference would have been spotted quickly.

11.7.6 Practice Tip 49: Automate the Execution of Build Code

Builds capture a lot of team knowledge. They can therefore grow to become significant code bases in their own right. The analytics coding chapter advised having one code file per data output. This makes it easy to find and test the program code that produced a particular dataset. In the case of build code, there will be one code file for every intermediate and interface dataset. This is potentially a large number of code files to open up and execute every time a build needs to be torn down and recreated. Executing build code should be quick and easy as this encourages the team to tear down and recreate builds often, as recommended in the previous tip. Additionally, being able to easily execute the entire build code base or some subset of it helps accelerate the development of the build.

In practice, this automation can be as sophisticated as available time and tooling permit. A simple shell script that recursively executes all code in a build folder is sufficient for some projects. Other projects use dedicated scheduling software that can execute code files in parallel where appropriate. Popular build tools from traditional software development such as Apache Ant (Moodie, 2012) can be appropriated for data analytics needs. Regardless of the approach however, a build cannot be automated if some of its data flow relies on technologies that cannot be executed programmatically from a script. This is another good reason why there is a Guerrilla Analytics principle to prefer program code over graphical tools.

11.7.7 Practice Tip 50: Embed Testing in the Build's Data Flow

Following the practice tips introduced so far, you should now have a version-controlled Data Build that tears down all datasets in its target namespace, and then automatically executes all the program code files necessary to produce its build datasets. As the build grows to incorporate team knowledge, it will become complex. This means you will need a quick way to know that its code has executed correctly and to report and summarize what the build code has done to the data. A critical component of the build process then is to create clearly labeled test datasets. These test scripts should execute during the build process so that a build can "fail fast" if it is not passing tests of its expected output.

Figure 41 shows the now familiar customer address example with the addition to two test datasets. The first test dataset checks that all postcodes in the cleaned address dataset are valid. The second test dataset checks that all customers from the raw data are still present in the final interface dataset of customer addresses.

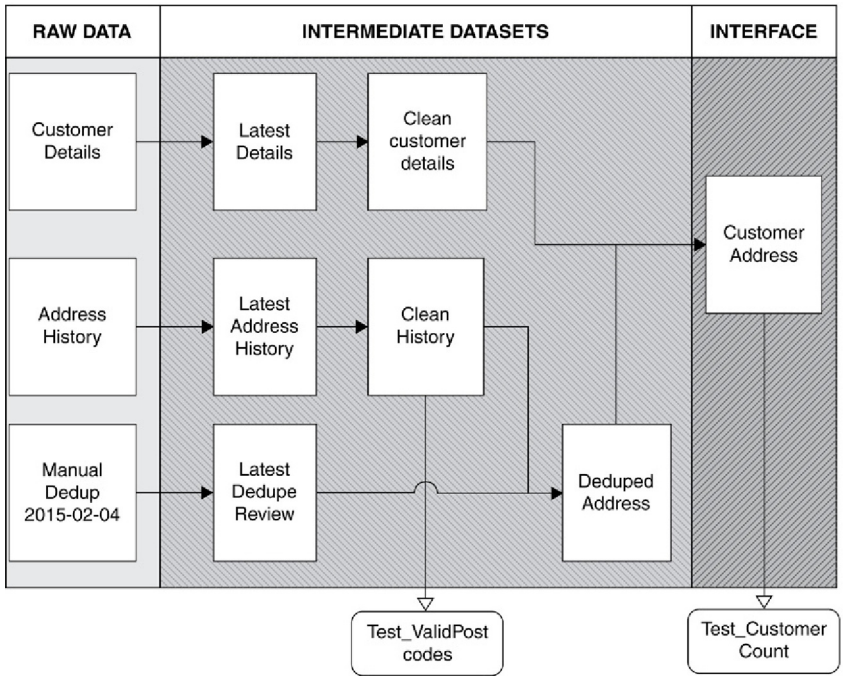

FIGURE 41 A build data flow with embedded tests

These are simple examples to illustrate the point. It is difficult to anticipate how analytics code will change the data and interact with the data flow. You should test to make sure your expectations about the build are being met. Testing is a huge area that is covered in its own part of the book. The point to emphasize here is that testing occurs during and beside the build process. There is no point in waiting until the end of a 2-hour build to discover that data records were corrupted in the very first data step.

11.7.8 Practice Tip 51: Embed Reporting in the Build

Much like testing, you would like to know what your build is doing to the data at key points in its data flow. That is, you would like to know the impact your build code has on data populations such as number of records, number of duplicates, date ranges, and so on at critical points in the build process. For example, a customer will often ask for the impact of applying certain business rules to the data. It makes sense then to incorporate this reporting of summary statistics into the Data Build process much as was done with testing. This has the following advantages.

- **Report on impact:** When a build completes, important reporting information about the build is already available. A team can easily say to their

customer that "the new population number is such a value". "The impact of the revised business rules is the following."

- **Quantify differences between build versions:** Having summary reporting information available as dedicated build datasets helps comparison of versions of a build. This makes it easy to demonstrate how critical project numbers might have changed between versions of the build.
- **Drive a modular build design:** When reporting is part of the build, it tends to drive a build design that is clear, modular, and traceable in terms of critical numbers derived from the data. If a report is needed on particular data steps in the build process then this drives the creation of intermediate datasets for easily creating those reports. This leads to a clearer and more modular build design that can be audited, reviewed, and tested without reworking intermediate datasets.

11.7.9 Practice Tip 52: Builds do not Produce Work Products

A danger when teams enthusiastically undertake builds is that they then try to create a build that anticipates all possible work product needs. In an ideal world, they imagine a build interface that has all possible datasets ready for use and work products simply export interface datasets for delivery to the customer. This is a mistake. The complexity of maintaining and running the build process will typically exceed any advantage of this design, even if all possible interface dataset needs could ever be anticipated anyway.

Work product and therefore customer needs should drive the build design not an attempt at anticipating all possible dataset needs. If patterns of repetition are seen in work products, then it is time to consider incorporating new functionality into the build interface.

11.7.10 Practice Tip 53: Create Documentation Datasets

As builds become more complex, it can become difficult for team members to remember exactly what every interface dataset represents and how it should be used? Where did that data field come from? Which build dataset do I go to for a particular data sample?

A traditional approach here would call for a data dictionary document to be maintained so that the team can look it up and understand how best to use build datasets. A Guerrilla Analytics project rarely has time for this approach. Analysts have to stop their analytics work to go off and find documentation and read and understand it. The documentation has to be maintained separately from the build code which risks its going out of date. It would be much better if the documentation sat right beside the code and data where analysts could easily find it when they need it. There is a simple Guerrilla Analytics solution here that creates a dataset that contains documentation. Table 2 is an example documentation dataset for some retail sales data that contains two interface datasets.

TABLE 2 An Example Documentation Dataset

Dataset	Purpose
PRODUCTS_SOLD	This dataset is a list of all products sold by day. Fields describe day of week, week number, year, and date.
PRODUCT_RANKING	This dataset lists all products sold and their rankings within various categories.

The advantages are as follows.

- **Version control:** The documentation is under version control (it is written in the build code after all).
- **Accessibility:** The documentation is easily accessible. It is right there in the interface datasets where the analysts are working and not buried in some folder on a file server.
- **Export for reporting:** The documentation itself can be queried and summarized if needed for reporting. With documentation datasets in the build interface, it is a simple matter to export these descriptions of the data and include them in work products when relevant.

Only simple written notes about datasets should be included in the build in this way. If build documentation really requires figures or software engineering tools then more traditional methods need to be used.

11.7.11 Practice Tip 54: Develop Layered Build Code

If you think about the build data flow, you can see how it takes code through several "layers." This was suggested in the structure of a Data Build illustrated in Figure 39. A build should always begin with pointers over raw data as discussed earlier. It almost always follows with data cleaning. Then there will be some type of joining or enrichment of various data sources. Then there will be a phase of applying business rules and doing analytics.

The intermediate datasets in the early "layers" have two important characteristics.

- **Low rate of change:** Unless data is refreshed or significant data quality issues are discovered, this lower layer build code does not need to change very often and the associated intermediate datasets therefore do not change.
- **Lower layers involve more data:** As the build data flow progresses, there may be some filtering of data so that the higher layers manipulate smaller datasets.

Looking at these characteristics of build layers, we see that lower layers are more time consuming to execute (there is more data), but change infrequently so rarely need to be re-executed. Architecting builds to have clearly defined layers

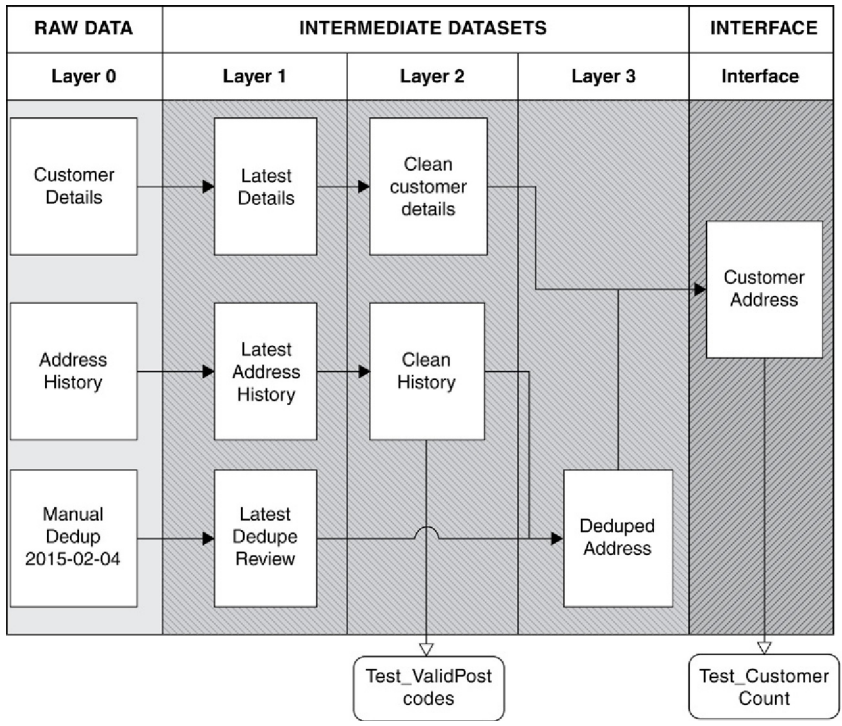

FIGURE 42 A layered approach to a build

allows these layers to be run independently rather than having to execute the entire build. This is very useful at development time as execution can focus on higher layers where the complex business logic is implemented. This of course does not mean that you should discontinue frequent tear down and rebuilds from scratch. Rather this makes code easier to maintain and to execute incrementally.

Figure 42 shows the customer address build with layers now identified.

- **Layer 0:** This is the raw data layer. This data is never modified.
- **Layer 1:** This is the pointer over the latest raw datasets. As discussed, this decouples raw data from the build data flow.
- **Layer 2:** This is the data-cleaning layer. This is the only place that cleaning rules are applied to the data.
- **Layer 3:** This is where business rules are applied. In the example shown, our business rule is the tagging of signed-off duplicate addresses.
- **Interface:** This layer contains interface datasets that are published to the team for use in their work products.

Note that more or fewer layers can be used as required but there should always be a Layer 0 for raw data and an interface layer. Layering can be included in the dataset namespaces to help identify datasets.

11.8 SERVICE BUILDS

Until now, this chapter has focused on Data Builds. Data Builds are convenience version-controlled *datasets* made available to the team so that work product development is simplified and consistency is facilitated.

Service Builds are the program code equivalent of Data Builds. They are common pieces of *code* made available to the whole team to avoid repetition and promote consistency. In a relational database, a Service Build may be a stored procedure or library loaded into the database. In another programming language, a Service Build may be a set of common macros or functions that the team can call on in their work product and Data Build code. Regardless of the team's chosen programming language, Service Builds are common code that the team should share. Since Services builds are stand-alone code, independent of data, they can be managed much like traditional program code using software engineering tools and processes with minor modifications for Guerrilla Analytics needs.

11.8.1 The Customer Address Problem Revisited

Earlier in this chapter, we discussed Data Builds in the context of a customer address-cleaning problem. Revisiting this problem, there are at least three areas that might benefit from a Service Build.

- In cleaning customer addresses and testing those addresses, it is useful to have some type of postcode or zip code validation. That is, rather than having complex pattern checking logic buried in the main Data Build code, it might be better to extract this logic into a validation function that could be used in several locations in the Data Build's cleaning layer. In addition, you might want to develop several functions for United Kingdom, United States, and other countries that each have their own address systems.
- The team might also notice that remembering the project's name cleaning rule of "remove all punctuation and numbers and upper case" is a little tiresome. They decide to also create a function for cleaning a given data field that captures this simple cleaning rule.
- Finally, the team decides to have date cleaning code. This code takes a given date field and steps through various dates, reformats them into a standard format, and then converts them to a date type in the DME. This saves having to remember a myriad of date conversion functions for every date type encountered.

These three function types for postcode validation, name cleaning, and date cleaning are example components of a Service Build. Over time, these functions will be modified by the team. For example, a postcode that had not been previously encountered may appear in a dataset refresh and the postcode validation function will have to be updated to accommodate this.

The customer address codebase now has two types of code files. It has the Data Build code files previously discussed. These manipulate data and apply business rules to create build datasets. Also, there is now Service Build code. This is version-controlled convenience code that can be called on from the rest of the code base. The following sections look at how best to create and manage Service Build code.

11.8.2 Practice Tip 55: Generate Service Builds with Version-controlled Code

Service Builds will undoubtedly evolve during the course of a project and between projects. New features will be required. Bugs will be uncovered and will have to be fixed. But what does this mean for the team's data provenance when the Service Build in question has already been put into "production" and been used in creating work products or Data Builds? Simply replacing the Service Build code will destroy the reproducibility of past work products and Data Builds.

The best approach is to version control all Service Build code. Older versions remain deployed and available to the team so that older work products and Data Builds can always be reproduced. Newer Service Builds get written and deployed as distinct new versions in the DME. Figure 43 illustrates a basic version control system for two Service Build functions and their deployment on the DME (in this case a relational database).

You can see that there have been two released versions of the postcode validation Service Build. These are numbered with a simple versioning scheme. Looking at the DME, you can see that both versions of the Service Build are loaded and deployed. Once again, this emphasizes the idea of maintaining a clear visual link between file system (where the code is developed) and the DME (where the code is deployed). If you find bugs in version 2 of the US postcode validation function say, or just want to better understand it, you know where to find its code quickly.

FIGURE 43 A basic version control system for a Service Build

Sample work product or Data Build code that uses this function could look like the following in SQL.

```
SELECT POSTCODES_002.VALIDATE_POSTCODE_US(PSCD) AS
PSCD_VALIDATED
FROM CUSTOMER_ADDRESS_002.ADDRESSES
```

As we saw with the Data Build, it is immediately clear which version of the Service Build for postcode validation is being used. Note also that this example shows a particular version of a Service Build being applied to a particular version of a Data Build.

11.8.3 Practice Tip 56: Embed the Service Build Version in Service Names

You will notice from the previous tip's example that the Service Build's version number is embedded in the DME relational schema name. In a Guerrilla Analytics project, you should always focus on maintaining data provenance with as little overhead as possible. A traditional software approach might deploy code so it overwrites previous versions of the function. No code dependent on this function would break and this would maintain backward compatibility of the code base. In the Guerrilla Analytics environment, your primary goal is maintaining data provenance. This means that all versions of code and data have to be maintained for reproducing historical results.

There are two advantages.

- By embedding the version somewhere in the Service Build's path, it is explicit which version of the Service Build is being used in a given piece of analytics code.
- By using a namespace for Service Build deployment, multiple versions of the Service Build can exist in the DME at the same time.

11.8.4 Practice Tip 57: Locate and Deploy Services Consistently

You are probably going to have multiple versions of several Service Builds in use at the same time on a project. The code files for these services need to be stored somewhere and the services need to be deployed somewhere. Without consistency in storage and deployment, it becomes difficult to maintain data provenance and services will accidentally be deployed to multiple locations.

- **Code location:** Locate all code libraries under one of the project's code folders. This lets the team find them easily and time is not wasted on conflicting code versions.

- **DME deployment location:** Deploy all services to one easily located place in the DME so the team can quickly determine what is available in the DME and what might be missing. Having to search the DME to determine whether a function has been deployed or not is time consuming.

11.8.5 Practice Tip 58: Tear Down and Rebuild Services

For development efficiency and ease of deployment, Services builds should have a method to tear down everything in their target deployment destination. Similar to Data Builds, there is a risk that incorrect versions of services are left lying around in the DME and that a team member then uses them on the assumption that they are correct. The first step in any Service Build process should be to tear down everything in its deployment destination location. In the relational database example already illustrated in Figure 43, the postcode Service Build was being deployed into the schema named POSTCODES_002. The first step in this Service Build's code should be to delete all existing functions within this schema. That way, only the services defined in the latest version of the Service Build code get deployed to the DME and no legacy functions are left lying around in the location POSTCODES_002.

11.8.6 Practice Tip 59: Automate Service Deployment

Similar to Data Builds, Service Builds may involve many program code files. In the postcode validation example, there are two code files to define the USA and UK validation functions. You would also need a preparatory code file to tear down any existing functions in the deployment schema as discussed in the previous tip. For other programming languages and analytics DMEs, some code files may have to go through preparatory steps such as compilation or packaging. The file or files for deployment then have to be moved to some deployment location.

This process should be automated with a build tool. Without automation, the manual deployment process is error prone. Developers have to remember the deployment location. Developers under time pressure will be tempted to edit functions as they are deployed on the DME rather doing development of the source code and redeploying and changes. As with Data Builds, a simple shell script may be sufficient for automating a Service Build.

11.8.7 Practice Tip 60: Services Builds do not Produce Work Products

As the name suggests, Service Builds provide services. That is, they provide convenience code that helps promote the team's efficiency and consistency. There may be a temptation to wrap up work product code in a Service Build because the work product is frequently requested by the customer. You would then have centrally deployed code with a very specific purpose. If

services are needed as part of a single work product, they should sit with that work product code rather than being shared with the team in a Service Build.

11.9 WHEN TO START A BUILD

When should you set up builds on your project? Do you wait until the team knowledge and data are complicated enough that they merit being stored in a build? How much time should be devoted to a build and its maintenance rather than direct delivery using work products?

You should start a build on the first day of a project, even for the simplest code and datasets. The following steps should be followed.

- **Build file:** Get a basic build framework in place with a build file for automation. This means that adding to the build will be easy should the requirement arise under tight timelines.
- **Assign a build master:** Assign at least one team member as the build master. Their job is to maintain the build and add new features to it as needed by the team.

11.10 WRAP UP

This chapter has discussed consolidating knowledge in Data Builds and Service Builds. Having read this chapter you should now know the following.

- **Why both Data Builds and Service Builds are needed:** Builds consolidate knowledge about the data and code functionality in a centralized location. This promotes consistency of work products that use builds and saves time redoing the same programming and data manipulation from scratch for every work product.
- **What causes data provenance to break:** Changes in code, libraries, and raw data affect any work products that use that code, library, or data. Controlling your evolving data knowledge involves controlling these three sources of provenance breakage.
- **Definition of a Data Build:** A Data Build is a collection of version-controlled convenience datasets that capture knowledge about the project data in a single central location for the convenience of the team.
- **Definition of a Service Build:** Service Builds are coded and version-controlled functionality that is made available to the team in a single central location. Typical Service Builds provide data-cleaning services and advanced functionality such as pattern matching and fuzzy matching.
- **How to version control Service Builds and Data Builds:** The key is to have a deployment namespace that includes the build name and version number. This allows multiple versions of the builds to exist at the same time in the DME.