Douglas Swanson

Boeing  10/16/2022

Genesis: Simulation Configuration Management

Contents

[Summary 2](#_Toc120424995)

[Introduction 3](#_Toc120424996)

[Reason 3](#_Toc120424997)

[Objectives 3](#_Toc120424998)

[Innovation and Uniqueness 3](#_Toc120424999)

[Scope 3](#_Toc120425000)

[Literature Review and Related Work 4](#_Toc120425001)

[Methodology and Design 5](#_Toc120425002)

[Overall Design 5](#_Toc120425003)

[The Simulation Environment 5](#_Toc120425004)

[Scenarios and Templates 6](#_Toc120425005)

[Alternative Designs 7](#_Toc120425006)

[Final Design and Implementation 7](#_Toc120425007)

[Data Storage 8](#_Toc120425008)

[Services 11](#_Toc120425009)

[Service Broker 12](#_Toc120425010)

[Front End 13](#_Toc120425011)

[Modeling, Simulation, and Analysis 15](#_Toc120425012)

[Logic Model 15](#_Toc120425013)

[Inputs 16](#_Toc120425014)

[Activities 16](#_Toc120425015)

[Outputs 16](#_Toc120425016)

[Testing 16](#_Toc120425017)

[Results 16](#_Toc120425018)

[Recommendations 16](#_Toc120425019)

[Ethical Implications 16](#_Toc120425020)

[Conclusion and Summary 17](#_Toc120425021)

[Acknowledgements 17](#_Toc120425022)

[Sources 18](#_Toc120425023)

[Appendix A: Professional Biography 19](#_Toc120425024)

# Summary

The GENESIS application is a proposed solution to the configuration of a Space Simulation currently in use by Boeing’s Virtual Warfare Center (VWC), which is a part of Boeing’s PhantomWorks division. The current problem with the simulation is that it is very difficult to create and configure a new simulation scenario in a reasonable amount of time. This application would allow the team to not only increase the speed in which they can develop a new scenario but also validate its inputs. This validation would give the team more confidence in the simulation when they run it either during an Operator In The Loop (OITL) event or for internal research and development studies. The GENSIS project will take the form of a flexible and high-speed framework that would allow the team to generate a new scenario form any network within the VWC. It would also allow the team’s software engineers to update and modify various modules to increase the capabilities of the software. All of this can be done using web technologies which would allow for containerization and deployment on any kind server. This project will be a dive into a method in which the team feels promising but still doesn’t have enough information to decide. So, in other words this a is test framework that may be flushed out further upon delivery or may even have some of it’s components completely redesigned. So this is a project to try new things and to see how some of these concepts work when actually implemented.

# Introduction

## Reason

The GENESIS simulation configuration management tool is a proposed solution to the slow turnaround and complex configuration of the Space Sim. The Space Sim is a simulation tool currently in use at the Boeing’s Virtual Warfare Center. This simulation allows the Space Team to run operator in the loop (OITL) training events and provides an environment to simulate and test future space architectures. The space team currently uses a Scenario Master Spreadsheet (SMS), along with a suite of accompanying software, to initialize scenarios and administer changes to objects in the simulation. The SMS is an Excel spreadsheet which currently gets the job done but, when more than one person starts editing a scenario it becomes an exercise in manually tracking changes.

## Objectives

In order to streamline the task of scenario generation, the GENESIS project should keep the following objectives in mind. Should be designed and built with modules in mind, like a modular framework. The inputs are key and should allow for validation. The framework should be well documented to allow for future developers to easily integrate new modules or fix current ones. The framework itself should have unit tests written. This framework should have some hooks written in to display the current scenario state to the user using proprietary tools like stellar. These tools would be rolled into the framework after it is delivered to the customer and would not be available until then. Should allow for the user to create a new scenario or load an existing one. Generate the appropriate schema from a configuration file. Populate the generated schema with data entered by the user. Display input forms so that a user can easily provide the data to be used during a scenario.

## Innovation and Uniqueness

GENESIS isn’t necessarily innovative, but it is a more focused configuration management tool. Most of the tools on the market seem to be aimed at any solution on any platform anywhere. This project is specifically designed to handle the setup and configuration of the Space Simulation for the VWC and that is it. It needs to be specific in order to provide it’s highly specialized users the flexibility and robustness they desire. It also needs to be flexible enough to change at the same speed as the simulation itself. So, this tool is a super specialized configuration management tool designed to run in the Virtual Warfare Center’s private networks, sometimes without access to the greater internet.

## Scope

The scope of this GENESIS project for this applied project course is as follows. The project should define the basic framework for the project. This framework should allow a user to generate a new scenario or load an existing scenario. Should allow the user to generate basic spacecraft data for this particular scenario through a webpage interface. Should generate the database schema and populate them with the provided data.

# Literature Review and Related Work

Currently the literature reviewed was used to get a general basis of what configuration management (CM), Service-Oriented Architecture (SOA), and Services are. Configuration Management is defined, by the Department of Defense as a system that “provides and orderly way to facilitate change...” (3). There are currently a few different software suites that perform software configuration management. One of the most well-known solutions would be Docker, which “takes away repetitive, mundane configuration tasks and is used throughout the development lifecycle for fast, easy and portable application development” (6). Docker’s direct competitor, Kubernetes is another software configuration management tool that handles much of the same activities using the same containerization model (8). On the far end of the software configuration spectrum there is Ansible which advertises itself as a full-service automation platform allowing its users to bring automation to their containers, security, applications, infrastructure, and cloud (1). On the opposite side, there is a tool called Config Cat which allows for a quick method of toggling features on individual applications (4).

A Service is “something useful a provider does for a consumer” (7) which can be anything ranging from a product on a shelf or something less tangible such as insurance or providing access to the internet. A service in regard to software is an application that has an interface that can accept requests from other sources, perform some action, and return a response. The action is what is desired by the source making the request. This interface is referred to as an application program interface (API) and it defines how to make a request of the service in question. There is a standard attached to this API called REST, which stands for Representational State Transfer. It outlines some rules that the API should follow. For instance, the API should use HTTP Methods explicitly, be stateless, outline its routes in URI like structure and accept and respond with data formatted using JSON (JavaScript Object Notation).

The services that directly deal with data manipulation are usually called CRUD services (11)(15), where CRUD stands for the actions it performs on the data. The actions are Create, Read, Update, and Destroy. These actions are mapped, if we are using REST principals(13), to specific HTTP requests. For instance, GET requests are mapped to Read actions, POST requests to Create actions, PUT requests to Update, and finally DELETE requests to Destroy actions.

A service broker is one method of providing the registry and discovery of services in applications designed to the SOA model. This application provides a central location in which the services running in the system can register themselves as available to receive requests(14). It also provides a method in which any component within the system can find out where any service within the system is running in order to send requests to it. Sometimes there is more to this application, especially in larger systems where multiples of any service may be running. This additional feature will provide the requesting entity with the address of the service with the least amount of work to do. This is called load balancing.

Now there are some software applications out there that accomplish something similar to what is trying to be done here. The first being the current solution used by the Space Team to configure the space simulation environment. They currently use a combination of excel spreadsheets and a service called FAST to convert the data from the spreadsheet into all of the records to run the simulation in the database. It works, but it has a lot of steps that can potentially introduce errors into the data. This current process is also time consuming and only a few people know all of the steps to fully complete it. Another alternative is using something like GitHub or GitLab. These options provide the features of the git software, branching, version control, change tracking, etc. This is done in a generic way to allow for may different types of software to make use of it, but it felt like we were going to have to make large logical leaps to fit our data into the git paradigm to accomplish our task. Now we do intend on tacking the framework itself within the git software but it would have been a chore to get our raw data to be tracked and would complicate the process of collecting and storing the data.

# Methodology and Design

## Overall Design

The Genesis application is designed with a Service Oriented Architecture, that way each of its components may be run on any number of networked machines within any company network. To do this each GENESIS module is written as a service that will have an API in which the other applications can request data from them. Each of these services will register itself in the Redis database using keys that expire after a specified amount of time. These expiring keys force the services to send keep alive messages to the database to receive requests from the rest of the system. It will also remove a service from the database should that database fail and be unable to remove itself. These services will have RESTful APIs in order to standardize the interactions with them throughout the system. These APIs will be well documented so that no one must go into the code to find out what the service features are. The user interfaces will be run off a single Node.js server that will both serve the page to the user as well as allow that page to access the rest of the application, wherever it may be running. These user interfaces will be written as react applications and will allow the user to perform CRUD actions on the GENESIS scenarios and templates and on the scenario specific data, depending on which templates that the scenario has been configured to use. Figure 1 below shows the most complex configuration of this framework, each service is running on a separate machine. This is the operational case taken into account when designing the main design for this application framework and any simpler case will work, like all of the components running on the same development machine.

## The Simulation Environment

The Space Simulation can run on a network of computers, some of these computers are virtual machines that run the databases as services and some are configured as terminals in which a user can interact with the simulation. GENESIS will be designed to run within this environment, hence using SOA and building the various components as services. This will grant the application the flexibility to run on any of the open machines within the network without necessarily being tied to a particular machine, except for the database services. This exception will actually be beneficial for the service discovery portion since all of the components will know where the Redis database is in order to register itself or find where the other services are running. One initial thought was to run all of these services in a Kubernetes cluster, but after further research, setting up and maintaining said cluster would take most of the semester. Kubernetes allows for a lot of users to access an application as it attempts to load balance by spinning up additional services and allows for the application to grow over time. This is a lot of overhead work for this particular application since there will only be a handful of users at any given time and the application doesn’t need to be running 24/7, only while a scenarios is being constructed. So, the user can run any of the GENESIS components through preconfigured batch command files at the time of use, on any of the machines they are currently working on. Regardless of this fact, GENESIS will be designed to run across any number of machines within the network or be placed into containers to run on Kubernetes in the future, should the need arise.

## Scenarios and Templates

In GENESIS we are dealing with generating and managing scenarios to be used within the space simulation environment. These scenarios will have a name, a description, a time that it was last updated, a unique key, a flag denoting if the scenario tables have been generated, and a list of templates that define the scenario's constituent modules. A module is an application or group of applications that model an aspect of the simulation environment. For instance, the Space Domain Awareness (SDA) module of the simulation is made up of 4 separate applications that allow players to schedule sensors to go look at space objects in order to determine their current orbits. Another module would be the Comm Effects (CommFx) module which handles all of the communication networks used by the satellites, which is comprised of 1 full application and some scripting in another. The templates will define the data required by the module and will let GENSIS know how to store, configure, retrieve, and validate this data. The template will configure the data table, within MySQL, through preprepared SQL statements that will define the tables in which the data is stored. The data model in the template will be used by the interface to develop a form, or series of forms, to collect the information from the user. In order to determine if this data is correct, the user interface will also use the defined data model to perform validation on the user's input. These templates will be directly related to each version of a particular module, and as that module changes so too will it's template. So, each template entry in the database will have a name, a version, a file path to its configuration files, and a brief description.

The current idea is to store the specifics of a template within a folder in the file system and create a record in the GENESIS scenario database pointing to that directory. The data model, SQL statements, and application configuration model will reside as flat files within this directory. One alternative to this would to use either the MySQL or Redis NoSQL database to store this data. Both of these databases would provide data security and an easy method of retrieving the records, through queries. The downside to using these databases would be trying to define these template models using the relational or key value store, depending on which is chosen, paradigms. The file system seems like a good way to start because it would allow for quick development and data flexibility of the templates as the rest of the GENESIS solution is created, at the cost of retrieval speed. Once GENESIS has been developed further and gone through a few iterations, this data may be migrated in order to gain the benefits of the databases.

In order to properly use these templates the implementation will answer the following questions. How will the data model be formatted for proper storage in the flat file system. Will there be dependencies used between templates? If so how will they be defined? How will the user interface forms be generated from the data model? How will they validate the data that will go into the data model? How will the CRUD service for this module be defined in such a way that it can handle different iterations of data models and thus different database schemas.

## Alternative Designs

Some alternative designs were considered for this project, such as a fully inclusive monolithic application that would use the Model View Controller pattern to provide the user interface, state management and database interactions rolled into a single application. But, this would have been a massive program which would be simple to deploy but a nightmare to update and maintain. Another design was a simpler version of the SOA architecture where a single service would handle all of the CRUD operations for the framework, a single UI application would be used, much like what was implemented, and it would drop the service broker. The broker would have been dropped because it would operate under the sense that the location of the service would be known and thus could be configured within the user interface before runtime. This was basically the fall back plan for this project just in case the more dynamic and flexible design detailed above failed to come together within the time frame of this course.

# Final Design and Implementation

The Genesis Configuration Management System was designed as a Service Oriented Architecture as explained in the methodology section above. In this architecture model, the Genesis system uses the services to perform the various actions upon the data submitted by the user. Currently, and in this implementation of the system they are only RESTful CRUD services that take in information and attempt to place them into the data stores. In order for the user to supply the data to the system there is a user interface designed to operate with the system services by supplying forms that take the user inputs validate them and send them off to the services, more on that later. In order for the front end UI to know what services are currently running and where, the system has implemented a service broker which allows the services to register themselves with the network address of the machine they are running on. The front end can query this service broker to find out if it can send it’s requests off to the desired service and where those requests need to be sent in order to reach those services. Most of this was covered in the methodology section above, we will go further into the details of each of these pieces of the system in the following sections. Here is an overarching diagram of how the Genesis Configuration Management System was built.

Diagram, schematic

Description automatically generated

Figure 1: GENESIS Application Interactions

## Data Storage

The system uses three types of data storage techniques to accomplish its goals. The first is a relational database to store both the scenario information and the data provided by the user to be used within each of these scenarios. The second is a flat file storage which is used to store the template data that is used to define each of the subsections of the simulation environment within the Genesis configuration management system. The third and final method of storage is a non-relational key value store which is used to track the running services and the network address of the machine they are running on.

To implement the main data store, a relational database called MySQL[] was used. This database was chosen because it is one of the leading relational databases on the market, it is free to use, it has exceptional data resiliency, can be installed on any operating system, and uses the widely know Structured Query Language (SQL) (). This is also the database that the Space Simulation uses to store its data while a simulation is running. Genesis makes use of relational databases for storing the scenario information it is suppose to be tracking as well as setting up the relational databases for the space simulation to run off of. A schema, or database, was created specifically for Genesis, called the *genesis\_scenarios*. In which three tables were created, one table for the scenario information itself, another table for the subsystem template information, and a third junction table that links the subsystem templates to the scenarios in a many-to-many relationship. The first two tables are standard data tables, which are defined below, but the third is interesting since we needed a way to link any number of defined subsystem templates to any number of scenarios that needed to use them. This method of doing things promotes not only the reuse of the template definitions within the table but reuse of the template definitions stored as flat files, more on that later. The two data tables used the data models below to build their tables.

|  |  |  |
| --- | --- | --- |
| Column | Data Type | Description |
| PKey | int | Unique identifier for this module template, will be automatically assigned to each record by the database |
| Name | varchar(25) | A human readable identifier for this template |
| Version | varchar(10) | The version of software this template belongs to |
| FilePath | Text | The directory file path to the folder containing the template’s files. |
| Description | Text | A more in depth description of the template and its purpose. |
| Deprecated | bool | A flag indicating that this template is old and has been replaced with a newer version. |

Table : Template Data Model

|  |  |  |
| --- | --- | --- |
| PKey | int | Unique identifier for this scenario, will be automatically assigned to each record by the database |
| Name | Varchar(25) | A human readable identifier for this scenario |
| Description | Text | An explanation of the purpose and or event that this scenario was used to support |
| Date Last Updated | Date Time | The last time that this scenario was updated by any user |
| Templates Used | Foreign Keys | The id’s of the templates used to create this scenario |

Table : Scenario Data Model

|  |  |  |
| --- | --- | --- |
| Column | Data Type | Description |
| PKey | int | Unique identifier for this scenario, will be automatically assigned to each record by the database. |
| Name | Varchar(25) | A human readable identifier for this scenario |
| Constellation | Varchar(25) | An explanation of the purpose and or event that this scenario was used to support |
| Rx | double | The starting x position of the vehicle |
| Ry | double | The starting y position of the vehicle |
| Rz | double | The starting z position of the vehicle |
| Vx | double | The starting x velocity of the vehicle |
| Vy | double | The starting y velocity of the vehicle |
| Vz | double | The starting z velocity of the vehicle |

Table : Spacecraft Module Spacecraft Data Model

One feature of MySQL is when defining a table, you must define a key for each item in the table but it also allows you to auto assign an id when a new record is added to the table which allows that database, the one source of truth on this data, to keep track of the next id number to assign to an incoming record but also guarantee that number is unique. This particular feature takes care of this tracking so that the services do not have to do it themselves.

In order to store more complicated data, such as the template data models and SQL queries, a flat file was chosen because of its flexibility for the types of data that can be placed within each “record”. The Genesis system is using the built in file directory system within the Windows 10 operating system as its flat file database. This offers a lot of flexibility, not only to the types of data that can be stored here but also allowing the data to be organized in any way. This flexibility does come at a cost though, its is relatively unsecure because anyone with access to the directory can go in a make changes to the data but it can be slow to store and retrieve the data. This was worth it due to the flexibility since the data that defines a subsystem template is complex and varies between subsystems. Genesis operates under the following assumptions, Each template directory will be placed in a single location, each template will have a directory matching its name, there will be a sub directory for each version of this template denoted as *v\_#.* Using these assumptions to add data to this flat file database allows for simple logic on the programming side in order to access this information.

Diagram

Description automatically generated

Figure : Template Flat File Database

In order to keep track of the services and their addresses a non-relational database called Redis was selected for its key value store and the ability to subscribe to changes to particular keys. The space sim also uses Redis in its environment for the publish and subscribe feature that Genesis could make use of in the future. We will talk more about the service broker that was built on top of this data store later. In order to be able to easily find the service data stored within the data store, the keys for the services took on the following shape, *genesis.broker.<service\_name>*, and store the Ip address and port they were running at as a string key.

|  |  |  |
| --- | --- | --- |
| Key | Data Type | Description |
| genesis.broker.[service name] | string | The Ip address, including the port of the service specified in the key name. Set to expire after 30 seconds. |

Table : Redis service key model

## Services

The services within the configuration management system are used to accomplish the CRUD actions with the relational data store and can be sent data through an application program interface which defines how outside applications can properly communicate with the service. The two services implemented in this project were build as NodeJS servers. A framework call Express was used within the NodeJS runtime environment to transform it into a webserver. This framework allows us to build routes into the server in which the users can access through URL calls, which allows us to follow the REST principals for this API. Once a request was made of the API, it would than take the information passed to it, parse any JSON data provided in the body, validate it, perform the requested action and respond to the user. Both of these service used a library called Winston to perform logging throughout this process. Any request, response, and error was logged so if something were to break during the operation of these services, than it could be easily tracked down later by a developer.

The first CRUD service built is called the Genesis Scenario Service and it handles all of the actions related to scenario and templates. It allows any program to request it’s API documentation by accessing the base route, *“/”,* on the machine it is running on. To follow the REST principals it is designed to be stateless, thus any information that it requires needs to come through the request itself or be stored in the database. For most of the routes all of the data required could be stored in the body of any given HTTP request. It will query the database itself when creating or updating any scenario or template because it will return the object stored in the database as part of the response body. The Genesis Scenario Service will takes care of coupling and decoupling of templates and scenarios. One unique feature about this service is that it will create the scenario database within MySQL when a scenario is created and will also generate the template tables within that database when templates are linked to the scenario. This is done as long as the template data store contains a Create SQL file for the template in question. All in all this is a very straight forward RESTful CRUD service without a bunch of added extras.

Text

Description automatically generated

Figure : Example of the API Documentation

The second CRUD service, called the Spacecraft Module Service, is directly linked to the Spacecraft template used to handle basic space object data within a simulation. This is a NodeJS server running with Express and Winston, and uses the same route paradigm as the Genesis Scenario Service, but is considerably more interesting. This service performs CRUD actions in a RESTful manner on the Spacecraft Template data requests sent to it. Since the template in question can have different versions the service needs to be dynamic enough to serve any defined version of the template at any time. This will allow for older versions of the template to be used should the team want to pick up and run or edit an old scenario from a past event. To do this the service has some custom middleware that does two things, it fetches the spacecraft data model from the template data storage location and passes it along the route. It also will choose the correct database in which to perform the requested action against based upon the name of the scenario provided with the request. As mentioned before, the request needs to provide both the directory path to the template being used and the name of the scenario in which the action is taking place. These are both provided as part of the header of the request since, GET requests typically are forbidden from having request bodies under HTTP protocol standards(). Other than these differences it is built very much like the Genesis Scenario Service.

## Service Broker

In the Genesis system we have services and servers can be ran on any machine in the network, but they still need to send requests to each other in order to work. In order to do this, they need to know where the services are running within the network. This is where a service broker comes in, it will know where the services are running and can be queried for any service that is running. This was done using the Redis database key value store. When a service starts up, in the Genesis System, it will report the location in which it is running on to the database by creating a key, following the pattern discussed above. When it creates this key, it will set it to expire after 30 seconds. This means that throughout the life of the service it will need to recreate this key in under 30 seconds of the last time in order to keep receiving requests, we call this a keep alive. This will cause the service to unregister should it run into any type of exception that brings the service down, thus causing the rest of the system to continue running without it. If this service is needed, the user will be informed to take action in order to bring the service back online, and ideally create a bug report for the failure. When another application wants the address of a service that is running, all that it needs to do it pull the list of keys matching the service name desired and pick one from the list, since if it is in the database it can be assumed to be running, and pull the address stored. It will than use this address as the base URL in which to make its requests of the service.

## Front End

The front end of the Genesis system was implemented as two parts, the first being a NodeJS server that would serve the second part, a React Single Page Application, or SPA. The server would run on some machine within the network and would allow the user to access the SPA when they connected through a web browser URL. It would also serve the template file data store through the same means. This would allow the React application access to each of the templates specific data models which it uses to dynamically generate the forms for the user to fill out. The server also provides a connection to the service broker, as discussed above. Now the interesting part is the React application itself, because it is where all of the logic for creating and managing the scenarios comes together. The react application comes up and allows the user to either create a new scenario or continue working on an existing one. If the user selects to create a new scenario, they are shown a form that allows them to enter the scenario details and to link subsystem templates to this new scenario. This form doubles and the template CRUD form where the user may Create new templates and edit or remove any existing template. Once the user is done creating the template they may submit the form which reaches out an uses the Genesis Scenario Service API to add the scenario and create the links between the templates and the scenario. Upon closer inspection, you would realize that there is no way to edit a scenario through the user interface, that is by design at this time. Editing a scenario by linking new templates or removing current templates would require major changes to the scenario databases. This was determined to be a possible feature in the future but would require some complex logic and more testing to incorporate properly.

Should the user decide to select an existing scenario they are taken to the scenario editing page where they can see the data associated with the scenario. This data is split into tabs, each representing a specific template. The user may both view the data in each tab as well as perform the CRUD actions on that data. All of the controls associated with the template are dynamically generated based upon the data model stored in the template’s flat file database location. The data model contains the fields in which to display to the user as well as the fields that the user needs to fill out in order to add another record to the scenario. In addition to the field, the validation type and limitations for the field are stored within this object. When the user interface control is rendered, it reaches out and dynamically imports the model in which to use in the code, which turns out is a super handy feature when trying to create dynamically generated forms and tables. To populate the table and submit the records, the UI reaches out and talks to the template service for its CRUD actions. In this case there is only one template, the spacecraft template and thus only one service to support it.

# Modeling, Simulation, and Analysis

## Logic Model

|  |  |
| --- | --- |
| Situation / Problem: | * The Space Simulation is currently too difficult to configure. * It takes too much time to create and validate a scenario. * There are too many errors introduced through the use of the current configuration process. |
| Inputs (Resources): | * Scenario Information such as name and description. * The list of simulation subsystems to be used in this simulation. * Customer data of the spacecraft to model within the environment for this scenario. |
| Activities: | * Defining a scenario by providing the asked for details * Inputting spacecraft information * Editing both scenario and spacecraft information |
| Outputs : | * Defined scenario:   + Name   + Description   + Subsystems * Validated and Stored Spacecraft Data ready to be used within the simulation environment |
| Outcomes (Impacts) |  |
| Short Term | * Quicker configuration of scenarios between events * Higher scenario confidence |
| Mid Term | * Simulation data become defined and understood * Higher simulation reliability * The ability to run more customer events generating more cash flow |
| Long term | * The ability to alter this tool to work with other simulation environments within the company |

Table : Genesis Scenario Management Logic Model

### Inputs

In order to define a basic scenario within the Space Simulation environment a few things must be known. The first being what is the goal of the scenario. Next would be what subsystems of our simulation are required to meet the goal of the scenario. From there we then need to know what information is required by these subsystems in order for them to function properly and successfully run the scenario.

### Activities

In order to define a scenario we need to give it a name and the description will note the goal of the scenario. After this information is provided, we will need to determine which of the subsystems will be required. This is done by looking at what each system provides the simulation and what information it produces during and after the scenario has completed. We can then cherry pick the modules we want and provide them with the data they require to operate.

### Outputs

In order for the space simulation to work properly with any given scenario, that scenarios data needs to be stored within

## Testing

The testing of this framework consisted of automated tests and manual data entry. In this case the automated tests were built to test the services and their static routes. These tests were designed to simulate valid and invalid entries to make sure that the obvious cases and some edge cases were covered in the error checking. These tests are run whenever the software changes to ensure that the inputs given provide the outputs expected. On the other side, the user interface was test through the process of entering the input values into the interface. After the inputs we entered and applied, the interface itself was checked to see if it had completed the task requested and delivered the expected result.

## Results

All of the test cases resulted in success, because the software was designed to accomplish

## Recommendations

# Ethical Implications

Throughout the design and implementation of the Genesis Framework engineering ethics was taken into consideration. This framework software is ethical, professional, and legal. It followed the guidelines outlined by the Boeing Company by not using any proprietary information or technology in the design or implementation of the software project, thus is legal. It is professional meaning that it treats the parties involved regardless of their background, ethnicity, religion, or gender. The software will not damage property or proprietary data, knowingly and has given credit where credit is due. Also, along these lines, the software has built in a manner to be transparent and uses open source software following the licenses agreements attached to that software.

# Conclusion

# Acknowledgements

During the course of this project I received help from the following people. Dr. Yinong Chen, for the initial design review and the idea to start with a Service Oriented Architecture. Will Sistar, for his insight into how this architecture could be incorporated into our simulation environment at Boeing. Eric Wolf for helping me walk through the Template Idea and assisting with researching multiple methods for dynamically generating forms based of a data model. Finally, my wife Amanda Swanson for watching our baby so that I had time to complete this project as well as supporting me throughout the semester.

# Sources

1. Ansible, Red Hat. “Ansible Is Simple It Automation.” Ansible Is Simple IT Automation, Red Hat, <https://www.ansible.com/>.
2. CHEN, YINONG. Service-Oriented Computing and System Integration: Software Iot Big Data and AI as Services. KENDALL HUNT, 2017.
3. “CM Overview.” Chief Information Officer U.S. Department of Defense, United States Department of Defense, <https://dodcio.defense.gov/Library/DoD-Architecture-Framework/dodaf20_cm_overview/>.
4. ConfigCat. “Configcat Feature Flag Service.” ConfigCat, ConfigCat, 1 Mar. 2022, <https://configcat.com/#product>.
5. Dikmans, Lonneke, and Ronald Van Luttikhuizen. *SOA Made Simple*. Packt Publishing, 2012.
6. “Home.” Docker, 24 Aug. 2022, <https://www.docker.com/>.
7. Jiang, Peishi, et al. “A Service-Oriented Architecture for Coupling Web Service Models Using the Basic Model Interface (BMI).” Environmental Modelling and Software, vol. 92, June 2017, pp. 107–118., <https://doi.org/https://doi.org/10.1016/j.envsoft.2017.01.021>.
8. “Production-Grade Container Orchestration.” Kubernetes, The Linux Foundation, <https://kubernetes.io/>.
9. Sen, Kaushik. “Top 10 Configuration Management Tools You Need to Know about: Upguard.” UpGuard, UpGuard, Inc, 12 May 2022, <https://www.upguard.com/blog/configuration-management-tools#toc-2>.
10. Whyte, Jennifer, et al. “Managing Change in the Delivery of Complex Projects: Configuration Management, Asset Information and ‘Big Data.’” International Journal of Project Management, vol. 34, no. 2, Feb. 2016, pp. 339–351., <https://doi.org/https://doi.org/10.1016/j.ijproman.2015.02.006>.
11. “Crud - MDN Web Docs Glossary: Definitions of Web-Related Terms: MDN.” MDN Web Docs Glossary: Definitions of Web-Related Terms | MDN, <https://developer.mozilla.org/en-US/docs/Glossary/CRUD>.
12. Pattinson, Tamara. “Relational vs Non-Relational Databases.” Relational vs Non-Relational Databases, 19 Oct. 2021, <https://www.pluralsight.com/blog/software-development/relational-vs-non-relational-databases>.
13. Rodriguez, Alex. “Introduction to RESTful Web Services.” IBM Developer, Nov. 2008, <https://developer.ibm.com/articles/ws-restful/>.
14. “SOA Service Registry/Repository and Service Discovery.” Coforge, <https://www.coforge.com/blog/soa-service-registry/repository-and-service-discovery>
15. “What Is Crud?” Codecademy, <https://www.codecademy.com/article/what-is-crud>.
16. Professional Biography