Automated Deployment of Distributed Simulation Environments Effectively Using Artificial Intelligence

Anup Raval, Gregory Tracy, Mark Schlottke & Zack Kiener
Dynamic Animation Systems, Inc.
Orlando, FL
{araval, gtracy, mschlottke, zkiener}@d-a-s.com

Jeremiah Long, Chris McGroarty & Christopher J. Metevier
US Army DEVCOM SC
Orlando, FL
{christopher.j.mcgroarty.civ, jeremiah.long1.civ, christopher.j.metevier.civ}@army.mil

Scott Gallant
SG LLC
Orlando, FL
Scott@EffectiveApplications.com

ABSTRACT

While the US Army is adopting a Modular Open Systems Approach (MOSA) to build its future Enterprise Modeling and Simulation (M&S), reaching that future is following an evolutionary approach that requires us to smartly leverage existing simulation systems that are often diverse and complex. The use of simulation varies in scenario, available simulation services, simulation architecture (e.g. middleware used), computing power available, and network infrastructure, among other considerations. As increasingly complex simulation systems are composed, deployment of these systems also increases in complexity and presents many challenges. The move towards a cloud-compliant solution eases the burden to some extent; however, simulation deployments still require careful planning by a team of experts, which typically requires effort for composition and deployment for each simulation exercise.

In this paper, we present our approach to optimize and automate the deployment of distributed simulation systems we have researched and developed as part of the Automated Simulation Management (ASM) project. We will detail our optimizer that takes into consideration different simulation architectures (i.e. single location, multiple sites, Modeling and Simulation as a Service (MSaaS)) and deployment platforms (i.e. local machines, cloud, and hybrid) to provide a solution for before and during run-time. Through a machine understanding of the scenario, service composition, (exercise) use case, and target environment, we automate the deployment of the simulation environment. The core of the solution is a genetic algorithm that processes data from a knowledge model and provides optimal deployment results. Finally, these results are processed by a Large Language Model (LLM)-based script generator to create an environment specific configuration. The result is an effective orchestration of simulation capabilities in order to meet the simulated goals balancing models/simulations, computing, and networking.

ABOUT THE AUTHORS

Anup Raval is a Software Development Team Lead with Dynamic Animation Systems. He has over 20 years of software design, architecture, and development experience. He is currently leading the Automated Simulation Management effort. His research interests involve simulation deployment, service composition, and Modeling and Simulation as Service (MSaaS). He has also worked with domain specific languages and generative programming in other research efforts.

Gregory Tracy is a Senior Software Engineer with Dynamic Animation Systems. He has worked extensively on the architecture and implementation of domain specific languages (DSLs) and software libraries—aiding simulation modeling and interoperability for the Department of Defense as well as robotic controls modeling for the For Inspiration and Recognition in Science and Technology (FIRST) Robotics Competition (FRC) (an international youth competitive robotics program). Mr. Tracy is a proud software & controls mentor and alumni of FRC robotics team 5816. He is currently working on a modular metaheuristic optimizer for the Automated Simulation Management effort as well several other supporting technologies for other research efforts we are currently engaged in.

Mark Schlottke is a Senior Software Engineer with Dynamic Animation Systems and has over 10 years in software design and development experience with 5 years in the Modeling and Simulation space. He is currently supporting the Automated Simulation Management effort. His interests include general applicability of emerging technologies, Game Engine Simulation applications, and Modeling and Simulation as Service (MSaaS).

Zack Kiener is a Junior Software Engineer with Dynamic Animation Systems. He is currently supporting the Automated Simulation Management effort. His interests include Machine Learning (ML) and Data Science. He received his Bachelor's in Computer Science from the University of Central Florida.

Scott Gallant is a Systems Architect with over 25 years of experience in distributed computing including United States Army Modeling & Simulation (M&S). Scott has led technical teams on distributed M&S programs for distributed software and System of Systems design, development, and execution management in support of technical assessments, data analysis, and experimentation. He is currently supporting the Simulation and Training Technology Center (STTC) as a contractor in the role of simulation architect and systems engineer across several projects including the work described herein.

Jeremiah Long is an Engineer for Advanced Simulation at the US Army Combat Capabilities Development Command, Soldier Center, Simulation and Training Technology Center (DEVCOM SC STTC). He has nearly five years of Modeling and Simulation experience under his belt. His research interests include the general applicability of emerging technologies, Artificial Intelligence applications, Game Engine Simulation applications, Systems Engineering, and universal Modeling & Simulation aspects. He assists in a variety of research efforts that integrate, develop, and demonstrate these technologies in a relevant Army and Department of Defense operational environment. He received his Master of Science from the University of Central Florida in Industrial Engineering, and his Bachelor of Science in Computer & Electrical Engineering Technology from Valencia College, both in Orlando, Florida.

Chris McGroarty is the Chief Engineer for Advanced Simulation at the US Army Combat Capabilities Development Command, Soldier Center, Simulation and Training Technology Center (DEVCOM SC STTC) with over 20 years of Modeling and Simulation experience. His research interests include distributed simulation, simulation architectures, applications of Artificial Intelligence technologies to simulation, novel computing architectures, innovative methods for user-simulation interaction, methodologies for making simulation more accessible by non-simulation experts, future simulation frameworks, and the application of video game industry technologies. He manages and leads a variety of research efforts that mature, integrate, and demonstrate these technologies in a relevant Army and Department of Defense context. He received his Master of Science and Bachelor of Science in Electrical Engineering from Drexel University in Philadelphia, Pennsylvania.

Christopher J. Metevier is the Chief of the Advanced Modeling and Simulation Branch at the US Army Combat Capabilities Development Command, Soldier Center, Simulation and Training Technology Center (DEVCOM SC STTC). He has over 35 years of experience with the Army and Navy in the Modeling and Simulation (M&S) field. His M&S experience extends across the acquisition lifecycle and includes the research, development, adaptation, integration, experimentation, test, and fielding of numerous simulation technologies and systems. He received his Master of Business Administration from Webster University and his Bachelor of Science in Electrical Engineering from the University of Central Florida.

Automated Deployment of Distributed Simulation Environments Effectively Using Artificial Intelligence

Anup Raval, Gregory Tracy, Mark Schlottke & Zack Kiener
Dynamic Animation Systems, Inc.
Orlando, FL
{araval, gtracy, mschlottke, zkiener}@d-a-s.com

Jeremiah Long, Chris McGroarty & Christopher J. Metevier
US Army DEVCOM SC
Orlando, FL
{christopher.j.mcgroarty.civ, jeremiah.long1.civ, christopher.j.metevier.civ}@army.mil

Scott Gallant
SG LLC
Orlando, FL
Scott@EffectiveApplications.com

BACKGROUND

Manually deploying simulation environments can be technically complex, require experienced simulation architecture professionals, and is suboptimal and error prone. We often throw hardware at the problem without fully understanding all the interdependencies of the simulation use case, scenarios, networking architecture, processing power, and the simulation modeling. It is also difficult to identify and correct errors at run-time that may cause entire execution runs to produce unusable data, resulting in days of lost effort. Additionally, software deployment technologies, cloud computing capabilities, and simulation middleware for network optimization technologies are rapidly changing and evolving which will require rapid changes to how simulations are deployed.

The Combat Capabilities Development Command (DEVCOM) Soldier Center (SC) Simulation & Training Technology Center (STTC) is focused on research and development towards the improvement of simulation development, integration, execution, and sustainment over time. We are in a unique position to advance the state-of-the-art, having executed several research projects over many years working on automated simulation composition, developing systems engineering representations of simulations, and deploying simulations across varying compute and networking resources.

We have created the Automated Simulation Management (ASM) research project to address the needs outlined above for optimal simulation deployment and error detection/correction using Artificial Intelligence (AI). By moving this complex optimal deployment determination from a manual, human-based engineering process to an algorithm, we can update the algorithm as cloud and deployment technologies evolve, benefitting the entire simulation community without requiring all engineers to be experts in the latest technologies. ASM has developed a knowledge model to encapsulate the machine understanding of a given simulation's purpose, scenario, services, and architectures necessary to support optimal deployment and configuration. We have also created a novel genetic algorithm used to find the optimal simulation deployment architecture based on the knowledge model data. Finally, we are using a Large Language Model (LLM) to create deployment scripts based on the output of the genetic algorithm.

This paper describes our research capturing the complexities of simulation environments, enabling us to apply AI to optimize the deployment, detect runtime errors, and eventually actively correct simulation executions. We provide a high-level view of the solution architecture and then dive deep into our simulation specific knowledge model, innovative genetic algorithm, and how we use an LLM to create deployment scripts based on the optimal configuration of each simulation service. We also provide our plans for this project going forward including self-healing of simulation executions.

INITIAL RESEARCH

At the start of the ASM project, we reviewed existing languages, frameworks, and libraries to identify relevant or similar efforts in the simulation automation area. Our goal was to identify any existing tools and technologies supporting machine understanding of scenarios, simulation services, and system deployment environments. Our efforts included a review of scenario standards, simulation systems standards, various middlewares, and existing distributed simulation systems. From this, we determined that there are currently no solutions directly addressing the issues of deployment optimization within the field of modeling and simulation. Having found no existing solutions, we shifted focus to a systems engineering decomposition of the simulation deployment process and considerations.

With the simulation deployment process and thoroughly broken down, we determined the technologies and concepts that we then designed, developed, and applied to formulate a solution.

SIMULATION DEPLOYMENT CONSIDERATIONS

ASM is focused on determining the required deployment architecture based on factors required for the simulation to execute, on optimizing the deployment for simulation performance across the entire scenario, and reducing the costs of scalable computing resources in a cloud implementation. In addition, ASM is also evaluating requirements to enable automation of the optimized deployments.

Deployment Optimization

Major factors affecting simulation deployment decisions include the scenario definition, the simulation use case, required simulation services, and the deployment environment (available network and hardware). Each of these factors provide input to, and present challenges for, simulation engineers to define and deploy the optimal configuration to meet the simulation users' needs.

Simulation Use Case

How a simulation will be used (and its compute and network requirements) primarily drives simulation deployment engineering. Large-scale use cases require many compute nodes and high-bandwidth, multi-site networks, while small-scale use cases may run on just a handful of lab computers. For example, large unit training exercises use combinations of Live, Virtual, and Constructive (LVC) simulations, whereas individual task training may be confined to a small lab or cloud partition. Although there are countless uses and configurations for simulations, we focus on these factors for optimal deployment: simulation scale (e.g., number of entities, terrain size, and number of servers/clients), execution timing (real-time vs. non-real-time), service connectivity (e.g., publish/subscribe and best effort vs. reliable), compute requirements (fidelity and resolution), and run-time events that drive dynamic compute and network loads.

Scenario Definition

The scenario contains information about entities, their locations, their capabilities, and the expected events that will occur, all of which affect the optimal deployment of simulation systems. Entity information including entity count, resolution (modeling detail), and their configuration are considered when designing hardware and network resources for the simulation execution. Expected events in the scenario drive the usage of the processing power and network bandwidth over time. The execution of each simulation service (processing and network loads) varies over time with each scenario.

Simulation Services

Applications or services used in the simulation environment have specific hardware and network requirements, specific communication middleware supported, and in some cases, specialized hardware requirements, such as Graphics Processing Units (GPUs) or Mixed Reality (MR) / Extended Reality (XR) hardware. Each service may support multiple configuration options and has its own minimum system requirements depending on the scale, modeling capabilities, or networking required to execute the use case and scenario over the prescribed middleware. It is important that we capture the hardware and networking requirements (per scale) for each simulation service that can be used in a simulation deployment. This information will be used by our algorithm to determine how services can interact with each other, the compute resources necessary to deploy each service, and the network infrastructure required based on the number of instances of each application.

Deployment Environment

Hardware and network resources available for deployment constrain the size and scale of the scenario and type of use cases supported. The available hardware and network infrastructure may span multiple sites and cloud resources. The algorithm needs to consider the hardware capabilities available to deploy applications (e.g. compute power, operating system, etc.), the networking connectivity between nodes (e.g. bandwidth and latency), and any specialized hardware available to match services to infrastructure when determining where to deploy each simulation service. When compute infrastructure is limited, deployment configurations can be easily determined because there are limited options. However, when there is a large number of available compute and network options, this algorithm will be most useful to determine the optimal deployment configuration.

Simulation engineers must consider the four factors mentioned above, each with more detailed parameters, when defining a deployment configuration for complex distributed simulation systems. This typically requires specific knowledge and experience, as well as significant time and effort to accomplish, and often, does not always result in the optimal use of available resources. This can lead to overallocation of some system resources while others are underused, presenting runtime performance issues, hard to detect errors, and time-consuming and tedious startup processes. In the case of cloud-deployed simulations, non-optimal deployments can significantly increase the cost to run the simulations.

Deployment Automation

In addition to the significant time, effort, and expertise needed to define an optimal simulation deployment, it also takes similar expertise, effort, and time to deploy and maintain this configuration on the target platforms for each simulation execution. To help reduce this effort and time, enable complex simulations to be deployed by a broader, less technical community of users, and prepare for a future capability for self-healing, ASM includes the generation of deployment scripts for the simulation services. These deployment scripts are manually executed initially, with a future goal of ASM to support automated execution of the deployment scripts.

ASM ARCHITECTURE

The ASM architecture (**Figure 1**) enables simulation engineers to invoke the deployment optimization capability through Application Programming Interfaces (APIs) or a Graphical User Interface (GUI). The Representational State Transfer (REST)-ful APIs provide an option for users wanting to integrate ASM directly with their current system controls, whereas the GUI enables users to invoke the deployment optimization capability separately and manually (Fielding, 2000). It was designed with three service layers focusing on specific parts of the problem domain, and a common set of services (service repository, knowledge model, and database) supporting all layers.

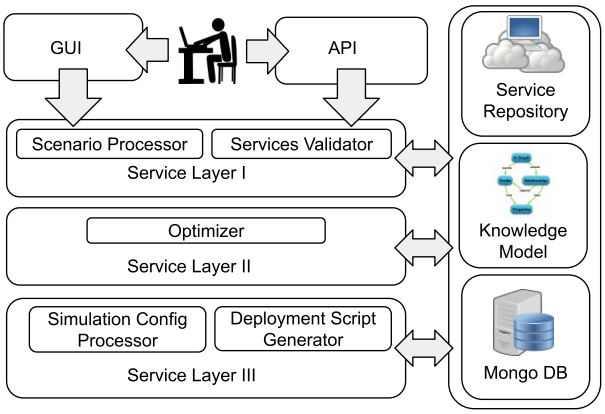

Figure 1: ASM Architecture

Service Layer I is responsible for processing and validating inputs and generating an output consisting of all the information used for deployment optimization. Users provide the scenario definition, target simulation service definitions and composition, and target deployment environment to Layer I using either the GUI or the APIs. ASM currently supports the Military Scenario Definition Language (MSDL) format and plans to support the Command and Control Systems to Simulation Systems Interoperation (C2SIM) format. Layer I parses the scenario definition to retrieve entity information, validates the service definitions against the service repository, and stores this data, combined with the target deployment environment information, as output to the MongoDB database.

Service Layer II retrieves the data stored in the database by Service Layer I, along with information from the knowledge model, and generates optimized deployment definitions. The generated deployment definition contains details about how applications can be optimally deployed in the given target environment.

Service Layer III is responsible for generating deployment scripts to enable deployment of services using the output of Service Layer II. It uses an LLM-based deployment script generator to generate executable scripts for deployment.

The Common Services group offers support for data storage, service configuration definitions in the Service Repository, and the Knowledge Model encapsulating different simulation architecture concepts.

SERVICE REPOSITORY

The service repository contains the deployment information required by the optimizer and the deployment scripts generator. The information stored in the service repository varies per service. Every service has the typical high-level information, such as its name and version(s). Other service-specific information includes a service's ports,

environment variables and their options, dependencies, specialized hardware required, minimum Central Processing Unit (CPU) and Random Access Memory (RAM) required for typical usage, and data persistence configurations (e.g. file paths, capacity in gigabytes, etc.).

The service repository consists of three parts: storage, web service, and user interface. Storage is done with an instance of MinIO (MinIO, Inc., n.d.). This was chosen for its ease of storage of various file types, its Amazon Simple Storage Service (S3) compatibility, and the flexibility to store information other than text, providing access to prefabricated scripts, configuration files, and other files that aid in deploying a service. The web server is developed using NodeJS, a runtime for JavaScript applications, and provides the API between the client and MinIO storage service (OpenJS Foundation, n.d.). This API allows us to provide a simpler and more consistent interface to access the stored information. It also provides a layer of abstraction should the need arise to replace the MinIO instance with an alternative storage service. The user interface is developed in VueJS, a JavaScript framework for building user interfaces, and is the primary path to adding and updating information to the service repository (Vue.js, n.d.).

KNOWLEDGE MODEL

The intent of the ASM knowledge model is to capture a machine understanding of scenarios, simulation components, and simulation configurations in order to support deployment optimization decisions. Initially, an ontology was used due to the ability to have semantically rich relationships of the data, the ability to reason on the data, the flexibility of the data and its relationships, and the availability of visualization tools to help better understand the relationship of the concepts (Noy & McGuinness, 2001). High level concepts related to hardware (e.g. CPU, RAM, GPU, etc.), networking (e.g. bandwidth, latency, etc.), application configuration (e.g. middleware usage, GPU usage, etc.), and deployment configuration (e.g. horizontal scaling for workload distribution, max latency for server and client, etc.) were captured within the ontology. However, during the eventual integration of the ontology within the ASM architecture, we determined the ontology tools and APIs came with unnecessary complexity and steep learning curves. To resolve these issues, we moved to an approach using the data model within the optimizer source code as a simple Domain Specific Language (DSL). This approach also provides an easier and more direct update path to add and test new concepts.

OPTIMIZER (SERVICE LAYER II)

With ASM, we are seeking to assign a potentially large and changing set of interdependent applications to heterogeneous compute resources—ranging from physical machines and virtual machines to container clusters. The optimizer must simultaneously satisfy hard constraints (e.g. CPU and memory capacities, redundancy requirements, or middleware compatibility) and optimize multiple objectives (resource utilization balance, communication latency, and operational cost). Given the vastness and ever-evolving nature of simulation software and computing hardware, a complete set of constraints and evaluation criteria for all of the aforementioned software and hardware will be nearly impossible to catalog or maintain for any one party. Therefore, we must ensure our approach can accommodate unpredictable future requirements, such as new resource types (e.g., GPUs), evolving communication patterns, novel affinity or anti-affinity rules, additional metrics, such as energy consumption or licensing limits, or other concepts we struggle to consider today.

Design Requirements

We recognized that the unknown constraints of mapping application deployments to compute resources is an NP-hard problem, ruling out exhaustive enumeration for all but the most simplistic scales (Papadimitriou & Steiglitz, 1982). We therefore identified the need for a metaheuristic optimizer, and with the considerations of the problem space, we identified the following design constraints (Glover & Kochenberger, 2003):

- **Modularity**: The optimizer must remain extensible for new constraints, objectives, or modeling constructs can be added without rewriting core search logic. Additionally, we want the ability to replace the optimizer algorithm itself without changing existing constraint or objective modules, acknowledging the no-free-lunch theorem, which proves that no single algorithm is best for all scenarios (Wolpert & Macready, 1997).
- **Ergodicity**: While true optimality cannot be guaranteed, or even necessarily defined, it is much more difficult to suggest the algorithm can reach optimality without the ability to consider all solutions (Blum & Roli, 2003). Therefore, we require the ability to reach any describable deployment through repeated search steps and any optimizer should be able to continue searching near any solution representable in the entire solution space.

- **Unbounded Size**: There is no hard limit on how many deployments, connections, or configuration elements a valid solution may contain. Candidates can grow arbitrarily complex as needed to meet evolving requirements. The solution space and traversal mechanism of the optimizer need to accommodate this.
- **Determinism**: Even though determinism imposes constraints on parallel and distributed execution, it ensures reproducibility of results, supports consistent demonstrations, and enables deep introspection during testing.
- **Immutable Modeling**: Each candidate solution is treated as an immutable snapshot; once constructed, its state cannot change. All evaluation and selection steps operate on fresh copies or derived data, preventing side effects or inadvertent state drift. Modules must respect this immutability, returning new instances for any transformation and never altering existing solution objects.
- **Parallel & Distributed**: Given the unknowable complexity of the domain the optimizer must operate over, serious consideration should be given to supporting a parallel and distributed architecture for the optimizer's search to aid our research effort. As the optimizer reaches maturity, it is strongly preferable that the optimizer does not require significant compute to find optimal deployment configurations, but this is unknowable as of now.
- **Traversal Ignorance**: The optimizer core should treat a candidate solution as a black box and solution models should never need to specify how the optimizer should traverse or transition within the solution space. The optimizer should generically be able to traverse a diverse solution space without consideration for specific data types or structures.

Genetic Algorithm

"Genetic Algorithms (GAs) are adaptive heuristic search algorithms that provide solutions for optimization and search problems. The GA derives expression from the biological terminology of natural selection, crossover, and mutation. In fact, GAs simulate the processes of natural evolution. Due to their unique simplicity, GAs are applied to the search space to find optimal solutions to various problems in science and engineering." (Höschel & Lakshminarayanan, 2019)

After weighing the complexity and dynamic modeling needs of ASM's solution space, we chose to use a GA for our optimization algorithm for the following reasons (Pétrowski & Ben-Hamida, 2017):

- The use of fitness functions for core objective modeling are in line with the goal of providing limited restrictions on the ways optimization objectives can be implemented in software. Fitness function-based objectives limit GA lock-in given its use in a wide range of metaheuristic search algorithms (i.e. tabu or simulated annealing) (Tomar, Mamta, & Singh, 2023).
- The use of genetic operators on a genome provides a clear abstraction between the optimizer's search mechanisms and the solution modeling enabling modifications to optimization strategies without an impact to how solutions are created or modeled. As mentioned with fitness functions, lock-in is reduced as metaheuristic search algorithms that traverse the solution space through arbitrary, non-specialized, actions can be adapted to traverse any genome's mutation function (Talbi, 2009).
- Unlike some other optimization algorithms, GAs can continue to make gains given the lack of a search gradient (direction of steepest fitness increase) (Goldberg, 1989).
- The population-based nature of the algorithm enables large parallelism without leaking any concepts of concurrency into solution modeling, fitness evaluation, and genetic operators (Talbi, 2009).

Currently, this GA works by maintaining a population of candidate solutions, called survivors, which evolves through a process of repeated iterations of offspring generation, evaluation, and competitive selection. The complete cycle is detailed in **Figure 2** and proceeds as follows:

1. **Termination Check**: A custom criterion determines whether the search should halt, in which case the existing survivors are returned.
2. **Reproduction Planning**: Two counts are computed: the number of survivors to retain, the capacity, and the number of new offspring to produce.
3. **Offspring Generation**: New candidate genomes are created in parallel by applying the variation operators (mutation & crossover) to survivors or by synthesizing new genomes.
4. **Genome Mapping / Birthing**: The new offspring genomes are mapped to the solution space. In this context, mapping is the

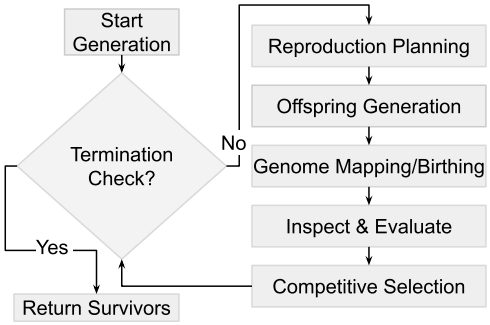

Figure 2: GA Process

process of converting the instructions to create a solution to the solution itself.

5. **Inspect & Evaluate**: The current population is assessed, emitting both numeric fitness values and metadata for external monitoring.
6. **Competitive Selection**: All survivors and offspring are pooled, their qualities reduced to a common metric, and the top individuals up to the capacity are chosen for the next generation.

It is important to note that given the current stage of our ongoing research, the quality of the GA used is of little importance. We are operating under the hypothesis that some variant of a GA, or GA augmented with secondary search strategies, is sufficient to produce solutions that are at least as optimal as that of professional simulation engineers in multiple orders of magnitude less time. Our primary focus is to show we can build an environment with robust modularity that can effectively model and optimize situations with high modeling complexity demonstrating capability beyond the examples reduced to their simplest form, a commonality (for good reason) in academic research that studies the topic of metaheuristics (Fu, 2002).

Genome (Genome Tree)

Many GAs use fixed-length bit-strings, which are efficient for mutation and crossover but unable to achieve ergodicity when the solution space is infinite, as is ASM's case (Whitley, 1994). Dynamic length strings can restore ergodicity in our case, but suffer from poor locality and potentially costly crossover. Instead, we employ a tree genome: each node holds a fixed-size bit-string and any number of children, allowing unbounded depth and size. This preserves information locality (mutations stay local) and enables a simple, efficient crossover. The rules governing the genome tree structure are:

- Each node holds a 64-bit unsigned value.
- Nodes may have zero or more child nodes attached.
- No two sibling nodes may share the same value.
- Sibling nodes under the same parent are kept in ascending order by their numeric values.

We have made our genome variation operators as general as possible, de-emphasizing their "quality" in line with our GA refinement philosophy. The no-free-lunch theorem tells us that no single strategy works best everywhere; our operators are no exception. For example, our mutation operator falters in stack-based genetic programming when the genome is viewed as an expression tree (Perkis, 1994). Introducing depth-transition mutations might help, but when depth semantics vary widely, it only slows convergence. To support our modularity requirements, which are at odds with the no-free-lunch theorem, both crossover and mutation expose extension points for modules to contribute to. Below, we outline our base variation operators for traversing the genome tree.

Mutation

A mutation is the process of transforming a single genome in some manner. For the genome tree, the mutation process is depicted in **Figure 3**. All decisions are made through a Pseudo Random Number Generator (PRNG) and are biased based on a set of hyperparameters configurable by the modules or experimenters.

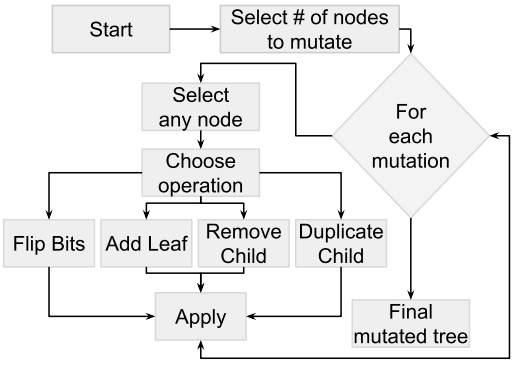

Figure 3: Mutation Process

Synthesis

Synthesis is the birth of new candidate solutions from minimal starting points. Beginning with a single root node, the process repeatedly applies randomized edits, such as bit flips, leaf insertions, child removals, and subtree duplications, to grow and diversify the structure.

Crossover

Crossover produces a new tree by weaving together the sorted children lists of two parents, preserving each tree's ordering and uniqueness rules without external weighting. The following is a more detailed description of the process:

- **Parallel Traversal**: Walk through both parents' child lists in ascending order of their identifiers.
- **Matching Siblings**: When both lists present the same identifier, either blend those two subtrees through a recursive merge or inherit one subtree in its entirety, chosen at random.
- **Unmatched Siblings**: When an identifier appears in only one parent's list, decide based on a simple random choice whether to include that child in the offspring or skip it, preserving potential novelty.

- **Exhaustion**: Once one parent's list is drained, remaining children from the other may still be considered for inclusion under the same random criteria.
- **Finalization**: After collection, duplicate entries are pruned, and the remaining children are re-sorted to ensure strict ascending order. The offspring's root identifier is then drawn at random from one of the two parents.

Birthing & Solution Modeling

To support extensibility and modularity, we designed the optimizer to use a minimalist, general solution representation: solutions are built by recursively interpreting the genome tree through a layered, modular property-binding mechanism. Thus, rather than embedding domain-specific constraints, the optimizer delegates to modules that bind genome segments to constructs like deployments, parameter values, or anything else a module author may need. Importantly, this modeling convention is provided but not enforced for module internals; each module remains free to derive nearly arbitrary Java Virtual Machine (JVM)-representable structures that best fit its domain's semantics. The use of mapping functions from each module to translate genome subsections to solutions cleanly decouples search mechanics from model interpretation, and allows modules to evolve their mapping and evaluation logic independently of one another.

Fitness Evaluation

Solutions are evaluated by collapsing multiple objectives into a single scalar fitness metric. Each objective emits not only a numeric value but also accompanying metadata, such as names, categories, and optional visualization hints, making it easy to inspect how each criterion contributes to overall fitness in dashboards or reports. New objectives are registered dynamically via the module system: each module declares its own scoring rules, and the core aggregator simply collects and integrates them without any changes to the optimizer's engine.

With our current approach, each core objective is normalized to a common scale and weighted to express its importance; the overall fitness is the sum of those weighted values. This weighted-sum formulation keeps prioritization explicit and interpretable.

The modular architecture allows us to transition to a true multi-objective search if desired in the future (e.g., Pareto-front optimization) with minimal disruption: modules would continue to supply normalized objectives, and a Pareto-front sampler would replace the simple weighted sum at the top level as the aggregation strategy.

Module System

We implemented the optimizer using a Kotlin-based modular architecture, running on the JVM (OpenJS Foundation, n.d.; Jetbrains, n.d.). We achieved modularity using JVM class-loading mechanisms, allowing dynamic discovery and integration of new modules packaged as standard JAR files. Additionally, we designed much of the optimizer's core public APIs with Kotlin's expressive DSL capabilities, enabling ASM module authors to concisely declare constraints, objectives, variation operators, visualizations, and metadata. As such, behavior and features that may be augmented using the module system include (Jetbrains, n.d.):

- **Constraints and Objectives**: Modules can define new evaluation criteria that integrate automatically into fitness calculations.
- **Variation Operators**: Modules may introduce custom genome operators or augment existing mutation and crossover strategies, seamlessly integrated into the evolutionary process.
- **Domain Extensions**: Support for adding new resource types, deployment rules, or modeling constructs through straightforward module definitions.
- **Inspection and Metadata**: Rich descriptive metadata and visualization hints can be attached, enhancing interpretability and user interaction.

Through this architecture, we ensure straightforward extensibility to enable future enhancements, such as new resource constraints, novel optimization criteria, or additional operational policies, to be introduced simply by adding or updating JAR modules without changes to the core optimization engine. While modules are expected to be developed and introduced to the optimizer by third parties, we have developed a number of modules which we believe to be highly likely to be important to most use cases, or would introduce concepts that would likely be referenced by multiple dependent modules. The sections to follow introduce four of the modules we felt met that criteria.

Core Module
This module is tightly coupled with the optimizer and module system itself and serves as the shared common declaration of core problem and solution space concepts, such as applications, deployment variants, computers, scenario, and deployments. Additionally, the core module declares numerous extension points for the concepts defined by the module itself as well as systems that are more rigidly defined in the optimizer's core. Those extension points enable the addition or modification to the concepts of top-level fitness evaluators, genome offspring operators, and numerous points within the optimizer's core User Interface (UI). This is the only module that the optimizer directly observes, and in turn, all additional modules must extend some extendable functionality declared by the core module to introduce new functionality themselves.

Compute Resources Module
This module encapsulates all definitions and evaluations related to computational assets and their utilization. Its responsibilities include:

- **Resource Type Definitions**: Declarative descriptors for each compute resource (e.g., CPU cores, memory, etc.), including default values, constraints, and visualizations.
- **Requirement Specification**: Binding of each deployment variant to resource requirement expressions derived from properties of each deployment, and the overall solution, seeded from the genome, and modeled as evaluable formulas.
- **Utilization Computation**: At evaluation time, the module computes both per-deployment and aggregate host-level usage. Different classes of computing hardware can implement specific logic for how deployments consume resources on the host hardware, useful for modeling resources that can't be shared or fragmentation on computing systems backed by a cluster of discrete computing hardware. For example, a Kubernetes cluster with 5 nodes, each with 10 CPU cores, deploying as many instances as possible of a container that consumes 3 cores, will at best have one remaining unused core on each system, 5 in total.
- **Resource-Based Fitness Metrics**: Dedicated scoring functions prevent over-utilization and minimize/balance peak resource utilization across computing systems, feeding into the overall fitness aggregation.

New resource types or utilization policies (e.g., GPUs, storage capacity, Input/Output (I/O) bandwidth) can be added simply by extending this module, without impacting core problem definitions or orthogonal modules.

Networking Module
The Networking module provides the foundational modeling constructs for expressing network constraints, communication patterns, and link-level characteristics among deployments. This module captures per-deployment constraints on the types of data transmitted, accepted connection technologies, latency tolerances, and bandwidth needs enabling the optimizer to reason about whether communication between components is viable, efficient, or optimal under varying deployment scenarios. This module introduces the following taxonomy of networking constraints:

- **Data Consumers**, which declare the types of data they require, the acceptable connection types, and latency and bandwidth thresholds.
- **Data Producers**, which expose data types and connection modes, allowing optimization over compatible consumer relationships.
- **Data Retransmitters**, which model routing or bridging entities that forward data unmodified, enforcing constraints on I/O link types and propagation delay.
- **Data Translators**, which receive one data type and produce another, modeling protocol translation or application-layer adaptation with detailed delay and bandwidth modeling.

The networking module also defines physical network topology primitives, such as switches, interfaces, and links each of which can have bandwidth and latency characteristics modeled independently. This allows deployment candidates to be evaluated not only on logical compatibility but also on physical feasibility, such as whether the latency budget between two endpoints can be met under the current routing.

Middleware Module
The middleware module introduces modeling constructs necessary for capturing and reasoning about middleware service dependencies, particularly those arising in distributed simulation frameworks. These middleware technologies often require complex coordination between applications that interact via shared object models, time management schemes, and standardized communication protocols. This module defines middleware connection points which declare the supported middleware platforms and associated requirements, such as object model compatibility, federation support, or network stack behavior. At runtime, deployments that depend on such services bind to these

points with the genome controlling the specific middleware configuration to be used. This module enables multi-party coordination over shared middleware infrastructure. It is designed to flexibly accommodate the heterogeneity of these systems without prescribing a one-size-fits-all abstraction layer. Gateways (bridges between middleware platforms and/or object models) are also supported as part of the deployment model. This allows the optimizer to discover configurations where communication is made possible across middleware boundaries, without requiring universal compatibility or explicit consideration from users.

Additionally, the module supports the modeling of deployments with customizable partitions of work (i.e. load) that can be distributed across deployments where each deployment binds a set of weights derived from the genome. This allows for dynamic load balancing and use case specific strategies to emerge as part of the optimization process. During evaluation, the module contributes constraints to the fitness system that penalize incoherent configurations, such as missing federation servers, incompatible transport protocols, or conflicting middleware platforms. These constraints are framed as numeric fitness contributions that are composable with other modules in the optimization pipeline. Future work may extend this module with support for time synchronization models (e.g., time management mechanisms) or middleware-specific performance modeling. Its current form, however, already provides an extensible foundation for modeling and optimizing middleware capabilities and limitations.

LLM-BASED DEPLOYMENT SCRIPT GENERATOR (SERVICE LAYER III)

The deployment script generator's purpose is to create scripts for deploying the optimized configuration solution. While a typical approach would be to create a codebase that can read in the optimal deployment solution and write the deployment scripts, this codebase would have to adapt to all simulation solutions and constantly be maintained in an ever-evolving simulation landscape. Therefore, we look at an alternative approach to this task using LLMs.

LLMs have recently performed remarkably well with Natural Language Processing (NLP) problems, such as translation and code generation (Chang, et al., 2023). LLMs are commonly prompted with instructions paired with context to generate a proper response (Wei, et al., 2021). For instance, with code generation, you may add an instruction with some criteria context and the LLM will generate code that meets that criteria. We extended this use case to deployment script generation by providing instruction paired with a configuration solution context to create the criteria for the scripts, and experimented with LLMs to determine their ability to reliably generate these scripts.

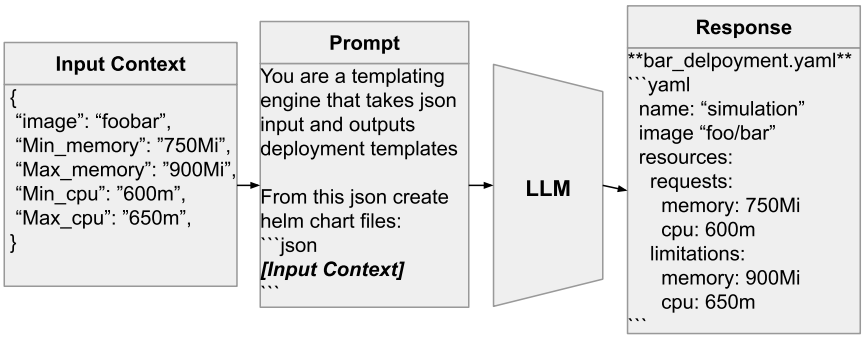

Figure 4: LLM Example Prompt/Context/Response

Experiment

We devised experimental data by creating our own JavaScript Object Notation (JSON) files representing multiple simple deployments with random configurations as our input context to mimic the optimizer solution (JSON.org, n.d.). Our goal for this test was to generate Kubernetes Helm charts as an output, with the ultimate goal to deploy them as an M&S as a Service (MSaaS instance) (kubernetes, n.d.; HELM, n.d.). This experiment was constrained to Kubernetes deployment and storage template YAML Ain't Markup Language (YAML) files within the Helm chart, and did not include other template files, such as those handling networking, dependencies, and secrets (Ben-Kiki, Evans, & Net, 2021).

For this task, we needed an open-sourced, pre-trained LLM with the capability to handle large amounts of text with multiple files. Mistral 7b instruct v0.3 has support for 32k tokens within its context window, enabling it to handle a significant amount of input text, thus making it a suitable candidate model (Hugging Face, n.d.). We used the JSON file data as input contexts into a prompt template with an instruction to construct the prompt. We provided these prompts to the LLM to generate respective outputs in a process known as zero-shot inferencing. We used prompt

engineering to manually prompt through different instructions with the same set of input contexts to generate different outputs. Our intention was to find outputs that met the criteria of files with file-path and content.

This process (depicted in **Figure 4**) produced some desired output, however, most of the outputs included hallucinations where the model failed to consistently produce a list of files or valid YAML files. This showed that our model required adjustment to improve the output.

Modifying LLM Output

In-Context Learning (ICL) is an output improvement method that uses sample prompts and responses concatenated with a new model prompt. This provides a demonstration to the model from which it can learn and adjust its output at inference time (Minaee, et al., 2024, p. 2). Each concatenated prompt and response are expected to be valid pairs that have been provided before the current prompt. For each k-number of prompt/response that is provided, it is considered a k-shot ICL inference. ICL has a significant impact on LLMs, as it is very effective for producing responses that follow a symbolic structure and pattern from the demonstrations (Dong, et al., 2023). Using this method does increase in inference time and GPU resource usage due to the increase in tokens from all the prior responses. This can become costly in deployments that need a significant number of files.

Supervised Fine-Tuning (SFT) is another method to adjust the model's output. It works by modifying the parameters of the model to favor the prompt and desired response provided, thus giving the model itself domain knowledge on the expected inputs and desired responses. Since our task is previously unseen to the model, the model will adjust to specialize in the task and conform the model to this instruction and type of context (Minaee, et al., 2024, p. 17). The way we fine-tune our model follows the Quantized Low-Rank Adaptation (QLoRA) approach, where mistral is quantized to 4 bits then it gets applied with Low-Rank Adaption (LoRA) prior to training. This allows for significant reductions in GPU memory, allowing room for memory spikes that occur during SFT, while providing near-negligible performance impact (Dettmers, Pagnoni, Holtzman, & Zettlemoyer, 2023; Hu, et al., 2021).

To evaluate these methods, we created a dataset of the desired responses following the previous output structure. This dataset contained 10,000 input contexts and responses, and a separately stored prompt template, enabling us to again follow the process in **Figure 4**. The dataset was created by making random JSON files, running those through a script to generate a known valid response, and then pairing them with the verified response. The dataset was split so that SFT train split was 80% and the test split (previously unseen by the model and used for evaluation) was 20%. We created two additional data pairs separately for a 2-shot ICL demonstration.

Evaluation

We used the test split of the dataset on benchmarks to give us a score based on the quality of the model's responses. Bilingual Evaluation Understudy (BLEU), Recall-Oriented Understudy for Gisting Evaluation - Longest Common Subsequence (LCS) (ROUGE-L), Metric for Evaluation of Translation with Explicit ORdering (METEOR) and CHaRacter-level F-score (chrF) are all text comparison benchmarks that generate a statistical score on how well the predicted values match each of their respective truth values, including the level of hallucination the model might be producing from the desired response (Papineni, Roukos, Ward, & Zhu, 2002; Lin, 2004; Lavie & Agarwal, 2007; Popovic, 2015).

Table 1: Benchmark scores for the LLM test split outputs

Model, Prompting Technique	Benchmarks (Higher is better)			
	BLEU (0-1)	**ROUGE-L (0-1)**	**METEOR (0-1)**	**ChrF (0-100)**
Base Model, Zero-shot	0.11	0.21	0.20	26.7
Base Model, 2-shot ICL	0.80	0.84	0.84	92.7
SFT Model, Zero-shot	0.76	0.73	0.67	80.76
SFT Model, 2-shot ICL	**0.91**	**0.94**	**0.91**	**97.2**

Table 1 shows the results from our inference run with the test split dataset on the Base and SFT-tuned mistral models. Each were prompted with zero-shot inference, where no modification data was provided with the prompt, and 2-shot ICL inference, where 2 valid examples are included in the prompt. ICL alone shows the greatest impact on performance on all scores while SFT shows significant performance improvements with both zero-shot and 2-shot ICL on all metrics. We found that base models are not familiar with the generation of deployment scripts and produce poor results. However, ICL is most effective at adjusting the model output, and SFT provided a superior model that

independently improves performance on all scores. We find that it is feasible for LLMs to reliably generate deployment scripts given the appropriate input modification.

Future Improvements

We are investigating adding a quantitative score component to the deployment script generation metrics to improve model evaluation. This metric would check each deployment for correct container images and values assigned to them, and provide a pass or fail score based on deployability and how well values matched from the solution configuration.

We have also separately created a deployment script template engine as an alternative script generation means. This template engine uses the Golang (Go) programming language, connects to the Service Repository for service details, and outputs simple Kubernetes YAML files for deployment. We plan to compare the output of this template engine method with the output of the LLM-based deployment script generator to further assess and adjust performance.

We plan to improve the data we are using to represent deployments. We will lessen the constraints on Kubernetes to also include other template files necessary for services, including networking, dependencies, and secrets. We also plan to account for deployment environments other than Kubernetes and include each of their respective configuration files. Lastly, we plan to add startup shell files to the data written to the directory. This will enable the LLM to specify the environment variables and command line arguments necessary in certain deployments.

We further plan to investigate methods, such as Retrieval Augmented Generation (RAG), to build upon ICL and extend the generalization of the LLM to include the increased input diversity required by deployments (Ram, et al., 2023). This method will retrieve external data to provide examples of deployment environments for modifying the model. This is also great for keeping the LLM up to date by having an updated catalog of example deployment scripts to which the LLM can refer.

Finally, we also want to investigate alternative LLM methods used in coding practice, such as adding a verification and reasoning step in the pipeline, to enable the LLM to then alter its output (Jiang, Wang, Shen, & Kim, 2024). This would add an additional verification step allowing us to test whether the deployment scripts are valid and can execute, or alternatively, providing error logs and potentially other resources as context for the LLM to make an improved output.

PATH FORWARD

Beyond the machine understanding of simulation needs and subsequent optimized deployment, ASM has two additional goals: simulation monitoring and automated self-healing of the simulation environment when problems are detected. We will be extending the ASM prototype to support these goals and will report on our progress in the future.

Simulation Monitoring

ASM's machine understanding of scenarios provides data for developing insights into the execution requirements, expected state of system services, and the hardware and network resources required for successful execution of each event in the scenario. We plan to use this information in conjunction with data gathered on the simulation services states, processing load of computers, and network utilization to predict execution issues that can then be addressed with the self-healing capability.

Self-Healing

The progression from simulation monitoring to automated self-healing is a large step in capability. Currently, a certain level of basic self-healing is supported by cloud orchestrators. Kubernetes can restart the container or redeploy pods upon failure. The ASM self-healing research will focus on issues that are simulation specific, less obvious to commercial tools, and can affect the outcome of a simulation event. We will investigate various methods to address detected simulation event issues with corrective action. We must be able to identify and understand patterns in data gathered by the simulation monitoring capability and map those with the simulation use case, scenario, and deployment to classify anomalies detected into actionable corrective actions. We must then determine how corrective actions can be executed to resolve issues during simulation executions without disrupting the simulation users or corrupting data. Finally, we must develop a means to automatically apply those corrections during simulation execution with limited or no intervention from the operators.

REFERENCES

Ben-Kiki, O., Evans, C., & Net, I. d. (2021). *YAML Ain't Markup Language (YAML™) version 1.2.* Retrieved from yaml.org: https://yaml.org/spec/1.2.2/

Blum, C., & Roli, A. (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys (CSUR)*, *35*, pp. 268-308.

Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., . . . Xie, X. (2023). A Survey on Evaluation of Large Language Models. *arXiv, 2307*.

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient Finetuning of Quantized LLMs. *arXiv, 2305*.

Dong, Q., Li, L., Dai, D., Zheng, C., Ma, J., Li, R., . . . Sui, Z. (2023). A Survey on In-context Learning. *arXiv, 2301*.

Fielding, R. T. (2000). Representational State Transfer (REST). In R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures (PhD).* (p. 82). University of California, Irvine.

Fu, M. C. (2002). Optimization for Simulation: Theory vs. Practice. *INFORMS Journal on Computing, 14*(3), 192-215.

Glover, F., & Kochenberger, G. A. (2003). Handbook of metaheuristics. *Springer, International Series in Operations Research & Management Science, 57*.

Goldberg, D. E. (1989). In *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman.

HELM. (n.d.). *The package manageer for Kubernetes*. Retrieved from https://helm.sh/

Hu, E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., . . . Chen, W. (2021). Low-Rank Adaptation of Large Language Models. *arXiv, 2106*.

Hugging Face. (n.d.). *Mistral-7B-Instruct-v0.3*. Retrieved from Hugging Face: https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3

Jetbrains. (n.d.). *Kotlin: Concise. Multiplatform. Fun.* Retrieved from Kotlin.org: https://kotlinlang.org/

Jetbrains. (n.d.). *Type-safe builders*. Retrieved from Kotlin: https://kotlinlang.org/docs/type-safe-builders.html

Jiang, J., Wang, F., Shen, J., & Kim, S. (2024). A Survey on Large Language Models for Code Generation., *2406*, pp. 33-34.

JSON.org. (n.d.). *Introducing JSON*. Retrieved from JSON.org: https://www.json.org/json-en.html

kubernetes. (n.d.). *Production-Grade Container Orchestration*. Retrieved from kubernetes: https://kubernetes.io/

Lavie, A., & Agarwal, A. (2007). METEOR. *Proceedings of the Second Workshop on Statistical Machine Translation*, (pp. 228-231).

Lin, C.-Y. (2004). ROUGE: A Package for Automatic Evaluation of summaries. *Proceedings of the Workshop on Text Summarization Branches Out*, (pp. 74-81).

Metaheuristics in combinatorial optimization: Overview and conceptual comparison. (2003). *Association for Computing Machinery, 35*(3), 268-308.

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., & Gao, J. (2024). Large Language Models: A Survey. *arXiv, 2402*, 2.

MinIO, Inc. (n.d.). *Hyperscale Object Store for AI*. Retrieved from MinIO: https://min.io/

Noy, N. F., & McGuinness, D. L. (2001). *Ontology Development 101: A Guide to Creating Your First Ontology.* Stanford: Stanford University.

OpenJS Foundation. (n.d.). *Run JavaScript Everywhere*. Retrieved from Nodejs: https://nodejs.org/en

Oracle. (n.d.). *JAR File Overview*. Retrieved from Java Documentation: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html

Papadimitriou, C., & Steiglitz, K. (1982). Combinatorial Optimization: Algorithms and Complexity. *IEEE Transactions on Acoustics, Speech, and Signal Processing, 32*.

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). BLEU: a method for automatic evaluation of machine translation. *ACL '02: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, (pp. 311-318).

Perkis, T. (1994). Stack-Based Genetic Programming. *Proceedings of the 1994 IEEE World Congress on Computational Intelligence* (pp. 148-153). Orlando: IEEE Press.

Pétrowski, A., & Ben-Hamida, S. (2017). In *Evolutionary algorithms* (p. 30). John Wiley & Sons.

Popovic, M. (2015). chrF: character n-gram F-score for automatic MT evaluation. *10th Workshop on Statistical Machine Translation*, (pp. 392-395).

Ram, O., Levine, Y., Dalmedigos, I., Muhlgay, D., Shashua, A., Leyton-Brown, K., & Shoham, Y. (2023). In-Context Retrieval-Augmented Language Models. *zrXiv*, *2302.*

Talbi, E.-G. (2009). *Metaheuristics: From Design to Implementation.* Wiley.

Tomar, V., Mamta, B., & Singh, P. (2023). Metaheuristic Algorithms for Optimization: A Brief Review. *International Conference on Recent Advances in Science and Engineering*, *59*, p. 238.

Vue.js. (n.d.). *The Progressive JavaScript Framework*. Retrieved from Vue.js: https://vuejs.org/

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., . . . Le, Q. V. (2021). Finetuned Language Models Are Zero-Shot Learners. *arXiv, 2109*.

Whitley, D. (1994). A genetic algorithm tutorial. *Stat Comput, 4*, 65-85.

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation, 1*, 67-82.