COGENT: Concurrent Generative Engineering Tooling

PDEng THESIS REPORT

COGENT: Concurrent Generative Engineering Tooling

Enabling Cross-Functional Teams in Architecture Design for Space Subsystems

Christopher A. O'Hara
October 2021
Department of Mathematics & Computer Science

PDEng SOFTWARE TECHNOLOGY

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

COGENT: Concurrent Generative Engineering Tooling

Enabling Cross-Functional Teams in Architecture Design for Space Subsystems

Christopher A. O'Hara

October 2021

Eindhoven University of Technology
Stan Ackermans Institute – Software Technology

PDEng Report: 2021/074

Confidentiality Status: Public

Partners

Siemens Digital Industries Software

Eindhoven University of Technology

Steering Group
Jonathan Menu
Mark van den Brand
Yanja Dajsuren

Date
October 2021

Composition of the Thesis Evaluation Committee:

Chair: Mark van den Brand

Members: Yanja Dajsuren

Yves Lemmens

Johnathan Menu

Tom Verhoeff

The design that is described in this report has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct.

Date October, 2021

Contact address

Eindhoven University of Technology
Department of Mathematics and Computer Science
Software Technology
MF 5.080 A
P.O. Box 513
NL-5600 MB
Eindhoven, The Netherlands
+31 402744334

Published by Eindhoven University of Technology

PDEng Report 2021/074

Abstract

System architecture design is a complex and complicated process. Systems, subsystems, and components must undergo a strict evaluation process detailing trade-offs, risks, benefits, and feasibility at the fringes of what is technologically possible. Poor architecture design leads to poor product performance, wasted resources, and in worst-case scenarios—fatalities caused by mission/product failure. Two upcoming domains seek to improve the generation, evaluation, and selection of system architecture configurations. These domains are generative engineering and concurrent engineering. Generative engineering allows for the automatic generation and evaluation of thousands of architecture configurations. Concurrent engineering is a methodology of subsystem design teams working collaboratively and simultaneously to create and select system architecture configurations. However, what had yet to be established was the value of combining the two domains. With Siemens SISW at the cutting-edge of technology solution development, this project sought to do precisely that in creating the Concurrent Generative Engineering Tooling (COGENT) platform. COGENT is a plugin solution architecture that enables cross-functional teams in automated system architecture generation in concurrent design facilities. A conceptual FireSat case study was explored, demonstrating COGENT capabilities such as enabling concurrent users, synchronized tool usage, centralized object storage, and connectivity to third-party software and/or user-defined features for space systems. COGENT is modular, extensible, and easy to integrate into any system development lifecycle. With COGENT, system designers can focus on their primary concerns, goals, and constraints. Using COGENT will allow system engineers, system architects, and subsystem designers to develop system architecture configurations at a fraction of the time and cost.

Keywords	Concurrent Engineering, Generative Engineering, Model-Based Systems Engineering, Space Systems Engineering, Systems Architecture, Artificial Intelligence
Preferred reference	COGENT, Concurrent Generative Engineering Tooling, Enabling Cross-Functional Teams in Architecture Design for Space Subsystems. Eindhoven University of Technology, PDEng Report 2021/074, October 2021.
Partnership	This project was supported by Eindhoven University of Technology and **Siemens Digital Industries Software**.

Foreword

Siemens Industry Software is a software tool vendor and supplier of engineering services for companies in the automotive, aerospace, and other advanced machine-building sectors. In recent years, it has initiated new activities in the domain of generative engineering: the automatic generation and evaluation of new system designs through computational techniques.

One of the challenges faced using generative engineering is accommodating the multidisciplinary and multistakeholder nature of large system and system-of-systems design. Indeed, while generative techniques will allow engineers to define and encode their design problem, the general evaluation and assessment process for generated system alternatives needs to be adapted to fit with these techniques. The decision process itself needs to allow designers to collaboratively analyze and score hundreds, thousands, or even millions of alternative solutions.

In this work, the COGENT project, Christopher O'Hara addresses one of the primary needs of the collaborative assessment of system designs: enabling designers to bring data objects together and to access the same objects in tools that allow for managing, tracking, and organizing them. The many faces data objects and system architectures can have often require enabling transformations between different data formats. At the same time, the envisioned solution requires a certain degree of flexibility to changing and new third-party tools.

The nature of this work – enabling to organize data objects – involves many interfaces to different tooling and process features. Properly scoping this work was one of the more difficult topics Christopher faced. This scoping required abstracting from front-end aspects (e.g., how a user interfaces the organization layer), procedural aspects (e.g., in which order a user prefers to use certain tools), as well as the collaborative decision support itself (e.g., decision algorithms). What remained is the essence of COGENT itself: a solution which Christopher termed "the COGENT plugin manager".

This final report describes the architecture of the plugin manager, and is the result of Christopher's relentless enthusiasm to deliver working and satisfying solutions. Even in a context which involved changing requirements and evolving insights, Christopher was eager to drive his tooling in an agile way. The result is a self-contained solution, which also provides value beyond the context of collaborative engineering. The insights on how to connect to versatile external tools, organized in a structured yet adaptive fashion, will steer further development activities of generative engineering at Siemens Industry Software.

Dr. Jonathan Menu, Research Engineering Manager

October 4th, 2021

Preface

This report describes the design, architecture, and proof-of-concept implementation of the COncurrent Generative Engineering Tooling (COGENT) Platform. COGENT acts as a plugin manager embedded in Siemens Simcenter Studio, concurrently connecting users to various software technologies via dynamic plugins.

This report describes the deployment of COGENT for a community-supported conceptual experiment (FireSat) within the context of the European Space Agency's (ESA) Concurrent Design Facility (CDF). FireSat is a complex satellite network system intended to discover forest fires. COGENT was verified and validated through use cases that demonstrate added value to concurrent engineering teams using generative engineering. Example Use Cases (UC) include:

UC1: Storing and analyzing architectural model metadata within graph databases

UC2: Versioning and tracking using experiment tracking software

UC3: Storing and managing data via on-premise cloud solutions

UC4: Analyzing feature co-factor dependence (correlation) and performing sentiment analysis via custom scripts

As a technology solution, COGENT enhances Siemens' generative engineering capabilities in the concurrent engineering domain. These enhancements directly translate to value for Siemens' clients as optimal solutions can be derived in a fraction of the time and cost by cross-functional teams. Additionally, COGENT integrates with third-party applications, providing great flexibility and customization at the user level without placing any dependencies on the primary software.

Christopher O'Hara conducted this project during a ten-month Professional Doctorate in Engineering (PDEng) in Software Technology (ST) graduation project under the direction of Siemens Digital Industries Software (SISW) and the Eindhoven University of Technology (TU/e).

The target audience of this report has a background in model-based system engineering utilizing system-level software solution architectures in the aerospace domain.

Christopher O'Hara

October 2021

Acknowledgements

No project is completed in isolation, regardless of a pandemic persisting for the majority of a two-year professional doctorate program. I am very grateful for the TU/e's Department of Software Technology as they have been very supportive throughout the program. I am highly appreciative of Yanja Dajsuren. She opened the doors for me to an endless number of possibilities. Before the program, I already knew what I wanted to do, but now I really know what I want to do and how to get there. I also want to thank the program's secretariat, Désirée van Oorschot, for all of her assistance. I could not have managed without either one of you.

I am also very grateful that Mark van den Brand was able to mentor my project. I know that he is a very occupied researcher and scientific director, so any time he can spare is a gift. He was very consistent with his feedback and guidance. I never needed to doubt whether or not I was on the right track, and if I did begin to deviate, he would steer me right back. It really is a privilege to work with him, and he is one of, if not the best, mentors for model-driven software engineering in the Netherlands.

From Siemens SISW, I really enjoyed working with Jonathan Menu. He is always supportive when providing guidance and direction. I think we have developed a great product together. I also want to thank Mike Nicolai for clearly specifying which technical questions would be the most meaningful for me to investigate. Furthermore, I would like to thank Johan Vanhuyse for always ensuring I had access to everything I needed while working with Simcenter Studio. SISW gave me every opportunity to have a meaningful and enjoyable project.

I am also appreciative to Tom Verhoeff and Yves Lemmens, external members of my Thesis Evaluation Committee, for taking the time to read my thesis, evaluate my project work, and listen to me present my efforts during the defense.

Finally, I would like to thank all of my supportive friends, family, colleagues, and peers over the years. Completing a doctorate is a struggle for the candidate and the people in their life. I have been away from home for four years now, yet everyone is still cheering me on. I feel very touched. Thank you for being a part of my journey, progress, and future. I still have many things to learn and skills to develop, so please bear with it a little longer.

Christopher O'Hara

October 2021

Executive Summary

Siemens Digital Industries Software (SISW) is a global leader in industrial automation, lifecycle management, electrical/mechanical design, and digital innovation. Technology-oriented companies and institutions regularly employ Siemens software within the automotive, aerospace, and intelligent manufacturing domains. Without exception, Siemens technology solutions blur the boundaries between various engineering and technology domains, allowing for effortless, multidisciplinary product development. Siemens SISW is currently developing a new technology solution for automatic system architecture generation called Simcenter Studio (SCS). At the start of this project, SCS was a single-user application. To enable cross-functional teams to engage in concurrent engineering, a methodology for simultaneous product development, SCS needed to be extended to a multi-user application.

We decided to create a plugin architecture to allow users to concurrently interact with models, access centralized storage solutions, and connect to third-party technologies. A plugin architecture was chosen since this pattern is modular, extensible, and easy to integrate into any system development lifecycle. This solution is known as the Concurrent Generative Engineering Tooling platform (COGENT). With COGENT, system designers can focus on their primary concerns, goals, and constraints. Using COGENT will allow system engineers, system architects, and subsystem designers to co-develop system architecture configurations at a fraction of the time and cost. A conceptual FireSat case study was explored, demonstrating COGENT capabilities such as enabling concurrent users, synchronized tool usage, centralized object storage, and connectivity to third-party software and/or user-defined features for space systems.

As a solution architecture, COGENT extends SCS from a single-user user application to a multi-user application with simultaneous tool access. Additionally, four use cases utilizing external or custom software were demonstrated for the FireSat case study, including graph databases, experiment tracking, centralized cloud storage, and user-defined modules. Furthermore, a centralized cloud solution was used to store objects, assets, metadata, and models in which end-users have concurrent and synchronous information. COGENT was designed as a high-level solution that can be extended to Industry 4.0 domains containing system-of-systems such as aerospace, automotive, industrial robotics, biomedical devices, and embedded Internet-of-Things.

Glossary

ACEL	Architecture-Centric Exploration Language
AI	Artificial Intelligence
AOCS	Attitude & Orbital Control System
AOP	Aspect-Oriented Programming
AQL	Arango Query Language
CAFCR	Customer Objectives, Application, Functional, Conceptual, and Realization
CDF	Concurrent Design Facility
CDP4	(RHEA) Concurrent Design Platform 4®
CDS	Computational Design Synthesis
CE	Concurrent Engineering
CFT	Component Fault Tree
CI/CD/CT	Continuous Integration, Continuous Development, Continuous Testing
COGENT	Concurrent Generative Engineering Tooling
ConOps	Concept of Operations
CPS	Cyber-Physical System
DevOps	Software Development & Information Technology Operations
DoDAF	Department of Defense Architecture Framework
DSE	Design Space Exploration
DSL	Domain Specific Language
DST	Domain Specific Tool
ECSS	European Cooperation for Space Standardization
EPS	Electrical Power System
ESA	European Space Agency
ESTEC	European Space Research and Technology Centre
ETL	Extract, Transform, Load
FoM	Figure of Merit
GCD	Generative Concurrent Design
GD	Generative Design
GDB	Graph Database
GE	Generative Engineering
HLR	High Level Requirement
HTIL	Human-in-the-Loop
KPI	Key Performance Indicator
LEO	Low-Earth Orbit
MBSA	Model-Based System Architecting
MBSE	Model-Based System Engineering

MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MGMT	Management
ML	Machine Learning
MOO	Multi-Objective Optimization
MVP	Minimum Viable Product
NASA	National Aeronautics and Space Administration
NoSQL	Not-Only SQL
NLP	Natural Language Processing
PDEng	Professional Doctorate in Engineering
PROP	Propulsion System
PSG	Project Steering Group
SAT	Boolean Satisfiability Problem
SCS	(Siemens) Simcenter Studio™
SEIM	Space Engineering Information Model
SoI	System-of-Interest
SoS	System-of-Systems
SISW	Siemens Digital Industries Software
SQL	Structured Query Language
ST	Software Technology
SYS	Systems
SysML	Systems Modeling Language
TCS	Thermal Control System
TDD	Test-Driven Development
TRL	Technology Readiness Level
TU/e	Eindhoven University of Technology
TQ	Technical Question
UC	Use Case
UI	User Interface
US	User Story
UML	Unified Modeling Language

List of Tables

Eindhoven University of Technology

List of Figures

Contents

1 Introduction

This chapter introduces the project context, the current Siemens Simcenter Studio software, the scope and goals of the project, and the report outline.

1.1 Project Context

Siemens Digital Industries Software (SISW) is a global leader in industrial automation, lifecycle management, electrical/mechanical design, and digital innovation. Technology-oriented companies and institutions regularly employ Siemens software within the automotive, aerospace, and intelligent manufacturing domains. Without exception, Siemens technology solutions blur the boundaries between various engineering and technology domains, allowing for effortless, multidisciplinary product development. These solutions are not only terrestrial, as Siemens supports aerospace partners—including the European Space Agency (ESA)—in space mission planning and design.

Space mission planning and design is a complex topic in the realm of System-of-Systems (SoS) [9]. Each experiment must go through a strict evaluation process detailing trade-offs, risks, benefits, and feasibility at the fringes of what is technologically possible. A majority of this design depends on architectural aspects within and between systems. As such, architecture generation is a critical component of the design phase. The early mission phases are considered crucial in many research and development groups, such as ESA's Concurrent Design Facility (CDF) [10]. With proper architecture design, potential problems are identified, mitigated, avoided, and/or resolved early during the design phase. In the CDF, representatives from different domains and subsystem teams co-locate and iteratively design complex SoS for intricate spacecraft, satellites, and robots. Utilizing Concurrent Engineering (CE) and iterative design, identifying optimal approaches to mission objectives is possible. Integrating state-of-the-art tools, techniques, and Model-Based System Engineering (MBSE) principles promotes mission—and experiment—success.

While CE improves quality, reduces costs, and saves time, it does not ensure the identification of an optimal architecture. Subsystem representatives must make numerous design decisions, trade-offs, and compromises during design sprints when deriving feasible system architectures. As such, CE needs to combine other methodologies for generating, filtering, and (down)selecting system architectures. One upcoming methodology is Generative Engineering (GE)—a computational design technique that automatically generates *all* feasible architectures. Furthermore, GE-created architectures are evaluated with quantitative Figure of Merit (FoM) values as constraints or goals.

1.1.1 Problem Description

While GE can impressively generate and evaluate thousands of potential architecture configurations for one experiment or product, the subsystem representatives still have finite time. Historically, the CDF ideates and evaluates between three and five architecture candidates for an experiment. However, GE alleviates the time spent in the initial architecture structure, interface verification, and system evaluation. After rapidly generating architecture configurations with GE, more of the designer's time is available for ranking, scoring, and analysis, with a confidence that a certain number of system architectures are "optimal" based on the mission objectives and operational requirements. Combining GE with CE may bring out the best of both worlds—*rapid system architecture generation with collaborative discussions leading to better system architecture selection while reducing costs and time during the conceptual phase.*

The problem statement is; *it has yet to be shown how generative engineering can be extended towards concurrent engineering for an integrated, multidisciplinary product team.* This project led to the creation of the Concurrent Generative Engineering Tooling (COGENT) proof-of-concept implementation. COGENT is based on the Concurrent Generative Engineering Activity at Siemens Digital Industries Software (SISW). This project aimed to: identify the optimal methods for storing data and assets, identify ways to track design decisions, enable cross-function team collaboration, and connect to user-specific, third-party tooling. We explored a conceptual FireSat space mission for validating these aspects via COGENT.

1.1.2 Simcenter Studio

Simcenter™ Studio (SCS) is an upcoming GE software application that enables automatic system architecture generation and evaluation of different system configurations during the early concept phase. SCS combines system simulation, optimal control methods, and reinforcement learning on top of a state-of-the-art machine learning and scientific computing stack to simulate and evaluate thousands of system architectures automatically. Combined with artificial intelligence (AI), engineers and data scientists can create user-defined procedures within computational notebooks for generative engineering.

System architecture generation begins with designers expressing their needs and constraints in an engineering-friendly formal model description. Next, AI guides the generation process systematically and reproducibly while avoiding typical design fixation challenges. In general, the early design phase allows designers and AI agents to collect critical design knowledge. With SCS, all feasible architectures can be identified, including novel solutions that human designers may not consider.

Computational *Notebooks* contain an integration of code, models, visualization, narrative text, and mathematical equations. The same environment contains all workflows, user-defined procedures, executable models, and documentation. For automatic controller generation, both a model-free approach (reinforcement learning) or a model-based approach (optimal control) can be used to create accurate and realistic trade studies. Multi-user, multi-attribute balancing utilizes these results via an intuitive web-based application called *Discover*. Finally, AI cross-filtering and recommender systems suggest, rank, and organize thousands of architectures based on the balanced attributes. The goal of SCS is to find the

best architecture configuration solution from the massive design space while tracking design decisions and enabling cross-functional team communication and discussions.

1.2 Scope and High-Level Requirements

Previously, SCS was used for GE in a single discipline as a single-user application. The project's scope is to take the generative engineering implementation of SCS and make it compatible with concurrent engineering for multiple users in multiple domains. Additionally, these users should be able to use third-party software within the SCS environment. Given the scope, four technical questions (TQ) were proposed:

TQ1: How to approach the scoring and ranking of system architecture configuration when combining generative engineering and concurrent engineering?

TQ2: What is the best way to interface with third-party software, and which third-party software is useful?

TQ3: What are the key aspects that can be implemented on a software level to help communicate, track, and justify design decisions?

TQ4: How should data be stored concerning structure, technology, and layering for a large number of missions, each containing assets, operational requirements, architectures, and models?

Based on the scope and TQs, six high-level requirements (HLR) are derived. These HLRs are provided below and are synonymous with "project goals."

HLR1: Extend SCS from a single-user to a multi-user application with simultaneous tool access

HLR2: Develop an end-to-end solution architecture, combining generative engineering with concurrent engineering

HLR3: Simulate the user ranking and scoring process, including syncing and consistency solutions

HLR4: Demonstrate potential use cases interfacing with external (third-party) software and tooling

HLR5: Implement methods for tracking, reporting, analysis, and discussing design decisions in cross-functional teams

HLR6: Develop a storage technology for storing and retrieving desired experiment (meta)data, objects, (meta)models, and assets

HLR4 derives from a challenge identified by Siemens SISW. The remaining HLRs were derived from the original project proposal or via requirements elicitation.

1.3 Report Outline

From henceforth, Chapter 2 discusses the problem analysis, Siemens-specific challenges, and the FireSat case study. Afterward, Chapter 3 describes the domain analysis together with background information and architecting methodologies. Next, Chapter 4 includes a product-based stakeholder analysis with use cases. Chapter 5 details the system requirements, while Chapter 6 describes the technical architectural designs. Subsequently, Chapter 7 details the system's verification and validation, with Chapter 8 providing the conclusions and recommendations. Project management aspects are in Appendix I.

2 Problem Analysis

This chapter first provides insight into general and space architecture challenges. Second, the Siemens-specific challenges that COGENT attempts to resolve are defined. Third, the FireSat mission is defined.

2.1 General & Space Architecture Challenges

This section details a few critical concepts related to general architecture design and space architecture design.

2.1.1 Wicked Complexity in Architectural Design

Architectural design is typically considered a *wicked* problem: "a problem that is difficult or impossible to solve because of incomplete, contradictory, and changing requirements that are often difficult to recognize" [11]. As systems are becoming more complex, understanding even a single Cyber-Physical System (CPS) is becoming more challenging for a single human designer [12]. Furthermore, (sub)systems are not designed by a single individual, as cross-functional teams containing designers and engineers from different domains (e.g., mechatronics engineering, electrical engineering) must work together when identifying constraints, goals, and interfaces. Without proper communication, an individual's interpretation of a component or interface becomes a barrier to an effective and efficient design. In complex CPS, including spacecraft, robots, and autonomous vehicles, multiple subsystem teams must work together to realize the final product—nearly always with conflicting goals. The integration of these decomposed CPS subsystems is contained in the SoS domain [9].

It is possible to conduct general architecture design concurrently, which adds another layer of complexity. Concurrent Engineering often requires computer modeling for Computer-Aided Design (CAD), Finite Element Analysis (FEA), and model simulation [13]. The results of these technologies should be exchanged efficiently, something that can be very difficult in practice. Stacking even another layer on top of this complexity is integrating Generative Engineering, with the automatic generation of possibly thousands of architecture/model configurations. Finding the best methods of data storage handling, knowledge transfer, ontology consensus, and cross-functional team communication is no simple feat.

2.1.2 Concept Phase Determines Life-Cycle Costs

In general, generative and concurrent engineering apply to the concept phase of a product or experiment. Additionally, CDFs explore the design space for experiments during the concept phase. The concept phase is one of the most crucial phases in a project or product. Decisions in the concept phase will impact cumulative life-cycle costs, cost reduction opportunities, and correctional costs [1]. Therefore, it is critical to identify the feasibility of designs and potential issues as early as possible. Fig. 2.1 illustrates the relationship between development phases and cost-related behavior. These aspects do not apply only to the system engineering domain, as poor initial design in software engineering can also be 1000x more expensive to fix a deployed product [14].

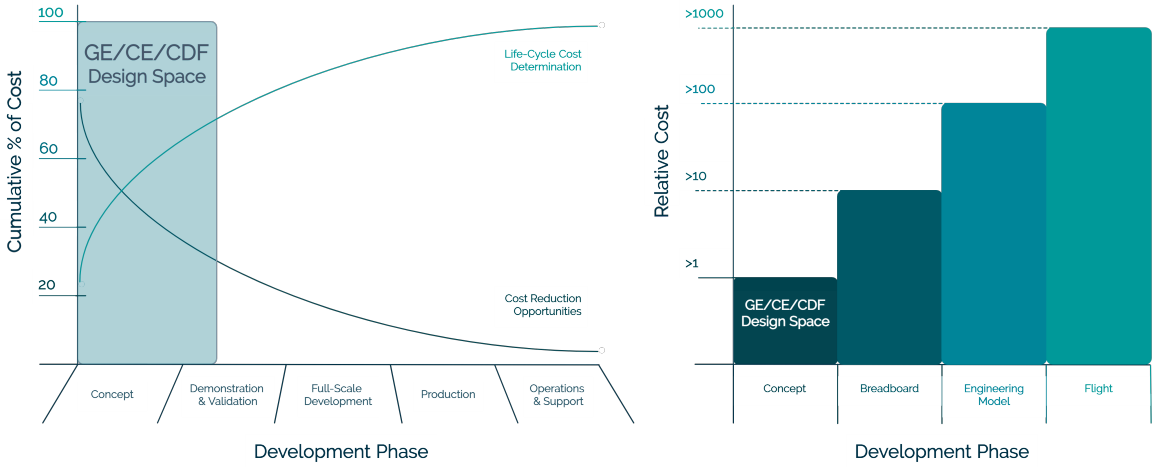

Figure 2.1: Cost vs. Development Phase. The figure on the left demonstrates that life-cycle cost determination increases and cost reduction opportunities decrease per phase. The figure on the right illustrates how drastically the relative cost of resolving issues increases ten-fold with each phase. Adapted from [1].

A significant motivation for combining GE and CE in the design of experiments and products at a CDF is to have more control and better results. Efficient and cost-effective system architectures minimize development costs and life-cycle resource usage. For safety-critical systems, finding the most reliable and safe architecture is a top requirement. For example, the 1986 Challenger Explosion directly cost NASA $3.2B and cut their funding by nearly half for the following two decades [15]. This operational failure was a result of poor communication between engineers and not identifying a suitable system architecture [16]. Space-based CDFs combining CE and GE, like the concurrent generative engineering activity, can mitigate issues that led to catastrophic failure by improving communication and system architecture generation.

2.2 Siemens-Specific Challenges

This section describes the Siemens-Specific challenges for COGENT to address.

2.2.1 Multi-User Extension

To apply GE in a CE environment, each user should contribute to system design from their individual workstations. Currently, SCS is a single-user, single-discipline technology. Extending SCS to a multi-user, multi-discipline implementation will enable CE. This extension is non-trivial, especially in the context of improving user communication, tracking design decisions, and handling multi-dimensional goals, constraints, and parameter values. Furthermore, this extension should demonstrate end-to-end behavior, starting from the model evaluation leading to a down-selected system architecture configuration.

2.2.2 Scoring & Ranking

Currently, in SCS, users can "like" an architecture configuration or configuration. Conversely, they are also able to "dislike" an architecture configuration. Liking and disliking architectures is effectively a filtering method for down-selecting potential architecture candidates. After the scoring process is complete, users will rank their top system architecture choices. This process is iterative, with multiple rounds of down-selecting occurring to achieve consensus between design teams. The COGENT project needs two processes: combining scores/likes for ranking and automatically generating a scoring/liking process. Additionally, a design challenge is storing and synchronizing the scores and ranks, along with assets and models. Finally, the recording and tracking of design decisions taken for each score/rank/like must be possible (motivation and justification).

2.2.3 Third-Party Software Integration

SCS is not meant to be used in a single domain or solve a single type of engineering challenge. Instead, SCS solves a variety of engineering challenges in any CPS domain. Software technologies have become broad in scope, ability, and usefulness. However, determining which technology is the most beneficial to use heavily depends on use cases. Instead of incorporating built-in alternatives to well-established third-party applications, it is more efficient to provide interfaces that allow users to interact with their preferred software stack. Interfacing with third-party software and which software is appropriate for a case study is part of a design challenge.

2.2.4 Case Study: FireSat

FireSat was, originally, a theoretical satellite network experiment. FireSat's concept is to use many interconnected satellites to identify or predict the occurrence of forest fires. Rapid identification of forest fires will allow fire response teams to mitigate the overall damage.

Predicting regions of high-risk (e.g., dry timber and low moisture) allows response teams to prevent forest fires from occurring by taking preemptive actions. The FireSat mission is an ubiquitous example applied in the development of space mission planning, design, and architecture [17]. Siemens has previously explored and expanded the FireSat case study for the concurrent generative engineering activity in generating system architectures [18]. With all of the challenges expressed, the next chapter will explore the domains.

3 Domain Analysis

This chapter details the domain analyses for GE, CE, and concurrent generative engineering. The domain models—and later metamodels—follow Model-Driven Engineering (MDE) techniques [19]. The last section introduces the CAFCR and DAARIUS architecting methodologies.

3.1 Generative Engineering

Generative Design (GD), also known as Computational Design Synthesis (CDS), is an iterative approach for implementing constraint-based or rule-based computational tools in the generation of potential design solutions [20]. GD usually occurs during the Design Space Exploration (DSE) phase of a project. Generative Engineering (GE) is an extension of GD that incorporates finite element method, topology optimization, and quantitative analysis to generate multiple versions of a part or architecture. A set of “best fit” designs is extracted from all possible designs from these generated versions. Using GE with optimization techniques enables designers to automatically generate and compare multiple designs to find an ideal “best-fit solution” using computational software. The designer can quickly iterate through thousands of possible optimized designs that meet trade-offs and requirements.

In SCS, an architecture is described based on its conceptual artifacts, connections, and constraints using a Domain Specific Language (DSL) known as the Architecture-Centric Exploration Language (ACEL). With ACEL, all feasible architectures can be generated as configurations with configurations. SCS implements and solves a Boolean Satisfiability Problem (SAT) which identifies (and classifies) all unique and duplicate architecture configurations, which can lead to thousands or even millions of possible architecture configurations depending on the number of components and complexity of the described system (subsystem example in Fig. 3.1).

3.2 Concurrent Engineering

Concurrent engineering (CE), also called Simultaneous Engineering, is a design methodology where teams work in parallel to realize complex systems and products, typically using design engineering or manufacturing engineering methods. CE consists of two primary principles. The first principle is that all product life-cycle phases require detailed consideration during the early design phases, including functionality, production, assembly, testing, main-

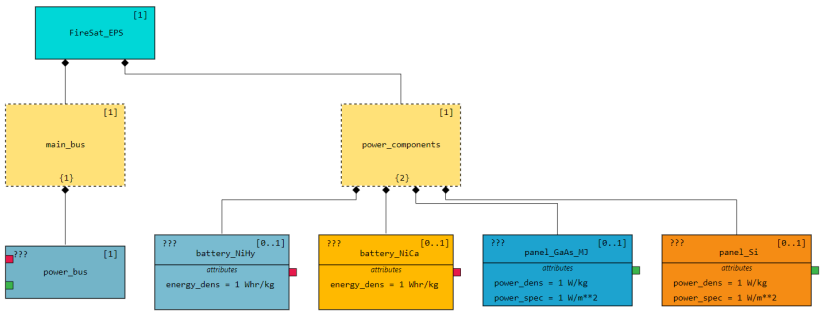

Figure 3.1: Generic FireSat subsystem architecture example generated with ACEL in SCS.

tenance, environmental impact, disposal, and recycling [13]. The second principle is that design activities occur simultaneously, i.e., concurrently, which significantly increases productivity and product quality during activities [21].

The benefits of CE at ESA/ESTEC for a typical pre-Phase A mission include: "a factor of four reduction in time, a factor of two reduction in cost, and an increased number of studies, and two parallel studies [22]."

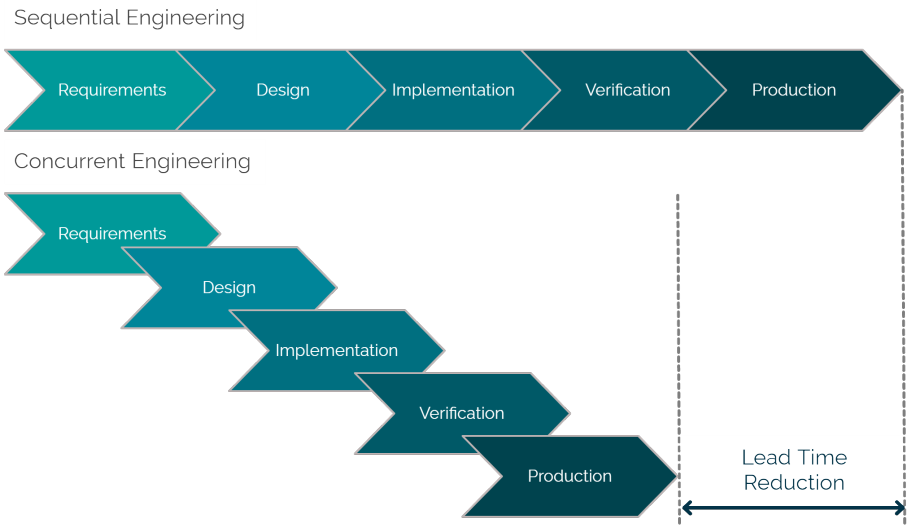

Figure 3.2: Concurrent Engineering process compared to Sequential Engineering showing Lead Time Reduction, adapted from [2].

3.2.1 Space Engineering Information Model

The ECSS-E-TM-10-25 - System Engineering - Engineering Design Model Data Exchange (CDF) is a Technical Memorandum under the E-10 "System engineering" branch in the ECSS series of standards, handbooks, and technical memoranda (further referred to as 10-25) [23].

The ontology-style 10-25 specifies standard data definitions in UML and defines the recommendations for model-based data exchange for the early phases of engineering design. The concurrent generative engineering activity and the COGENT project must comply with the 10-25(A) for creating a Space Engineering Information Model (SEIM) as a constraint.

The SEIM is a centralized model responsible for ensuring a consistent understanding of definitions and parameters for all designers working on the same space mission project. The SEIM stands as a "single source of truth" and is similar to an architecture framework. Instead of having backward traceability to the SEIM, COGENT assumes that any implemented quality attribute, data type/value, specification, or parameter was formally defined.

3.3 Concurrent Generative Engineering

This report sees concurrent generative engineering from two viewpoints: an independent domain and an Siemens' Activity. From the high-level domain view, concurrent generative engineering combines GE and CE, effectively allowing the generation of all possible architecture configurations to be made available for CE design teams. As an Siemens' Activity (which was the basis of COGENT), it is a Space SoS workflow regarding generating, evaluating, and discussing viable space architecture configurations. In this activity, Siemens works with industry partners to combine MBSE tools for architecting complex space systems [18]. Fig. 3.3 shows the interpreted concurrent generative engineering domain model and context used to formulate later COGENT domain models and metamodels.

3.4 Architecting Methodology

The primarily used frameworks for the project were CAFCR [3] and DAARIUS [4], with a combination of UML, SysML, and DoDAF [24] modeling methods depending on which modeling method is the most suitable for communicating information. These frameworks were chosen over Kruchten's "4+1" approach [25] since Kruchten's architectural views focus primarily on software development do not explicitly relate or incorporate parameters. Additionally, CAFCR and DAARIUS are compatible and designed precisely for creating systems like COGENT.

3.4.1 CAFCR

"CAFCR" is an acronym concatenating five system-architecting viewpoints: Customer Objectives, Application, Functional, Conceptual, and Realization. The CAFCR model assists system architects in rapid context switching between viewpoints in order to develop reliable, valuable, and usable products [26]. Using the CAFCR model iteratively and recursively bridges between relevant viewpoints and enhances this context switching throughout the system lifecycle (HLR2). Since developing COGENT centered around multidisciplinary and multi-domain teams creating complex CPS and SoS, the CAFCR framework is harmonious (HLR1) [3]. Fig. 3.4 shows an adapted CAFCR model.

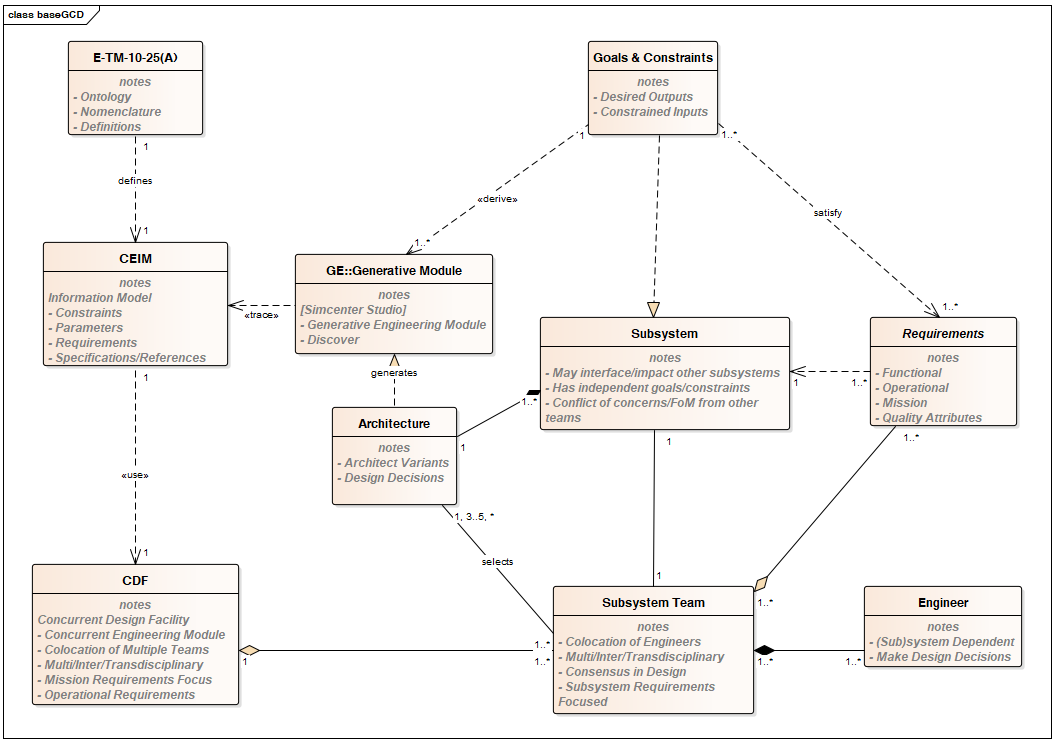

Figure 3.3: Interpreted domain model of Concurrent Generative Engineering within the context of COGENT.

3.4.2 Model-Based System Architecting

The DAARIUS methodology, formerly called Model-Based System Architecting (MBSA), is a scalable and structured system design methodology providing consistency and traceability for key design decisions in systems engineering for complex SoS and CPS (HLR5) [27]. DAARIUS works best when developing systems/products with limited knowledge, uncertain information, and lack clarity. Additionally, DAARIUS is based on and extends the CAFCR framework, focusing on decomposing subsystem team goals, constraints, and parameters [28]. The DAARIUS methodology guides on selecting which concepts to design when building an end-to-end system (HLR2), creating a system overview, and down-selecting the two or three best options (or architectures) [29]. Furthermore, DAARIUS uses relations to visualize trade-offs, assigns parameters and concerns to a specific domain architect/specialist, and considers quantitative and qualitative aspects across disciplines. Fig. 3.5 illustrates the DAARIUS methodology below. Since As DAARIUS meets both the needs of the COGENT project and the FireSat case study, it seems like the optimal methodology combined with CAFCR. DAARIUS and MBSA were used for requirements elicitation.

With the domain analyses and architecting methods complete, we can consider the stakeholders and their needs in more detail. The next chapter will describe the Customer Objectives Viewpoint with narrative use cases, user scenarios, and user stories. Additionally, we briefly describe and motivate the technology solutions for COGENT.

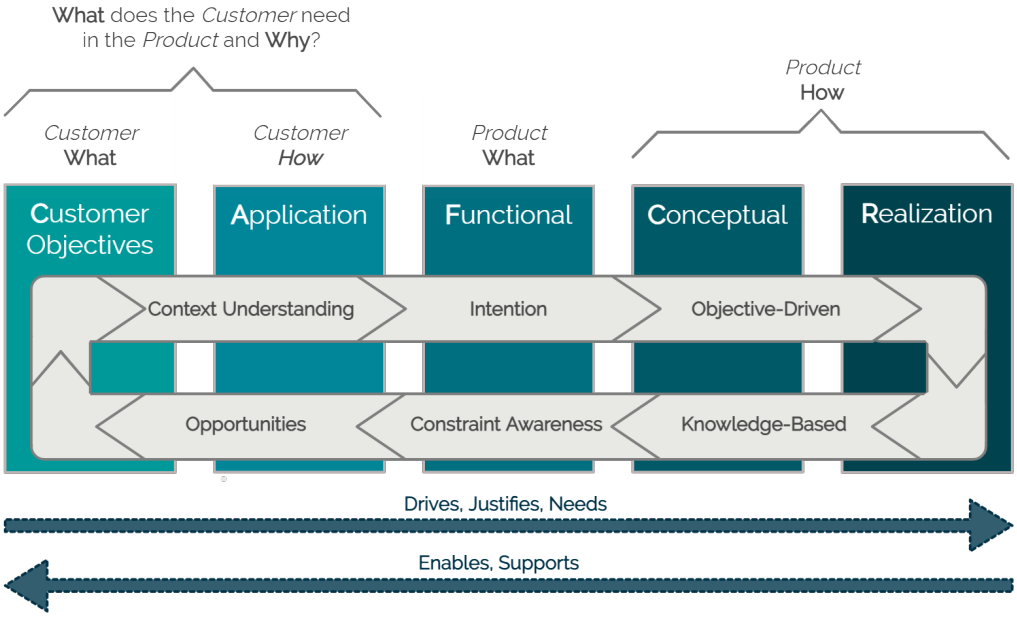

Figure 3.4: CAFCR, adapted from [3].

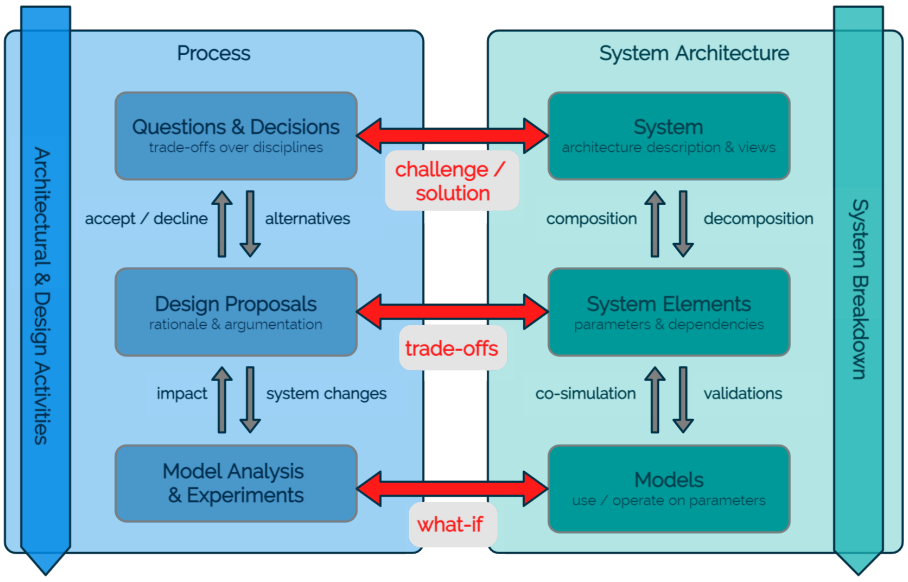

Figure 3.5: DAARIUS methodology overview, adapted from [4].

Eindhoven University of Technology

4 Stakeholder Analysis & Use Cases

This chapter is responsible for stakeholder analysis from the *Product Viewpoint* of the COGENT (for *Project Viewpoint* stakeholders, please see Sec. A.1). Next, Concurrent Design Facility (CDF) users and tasks are described, along with how COGENT is situated in a mission/product life cycle. Furthermore, four Use Cases (UC) derived from user scenarios are translated into user stories. Finally, this chapter includes the technological solutions description including graph databases, experiment tracking software, cloud storage, and user-created domain-specific tools.

4.1 Concurrent Design Facility

A Concurrent Design Facility (CDF) is a state-of-the-art facility where several transdisciplinary/multidisciplinary expert teams apply the concurrent engineering method in designing future space missions. CDFs contain a network of computers, multimedia devices, and software tools. It facilitates a fast and effective interaction of all disciplines involved, ensuring consistent and high-quality results in a much shorter time [6]. For ESA, The ESTEC CDF is the primary assessment center for future ESA space missions, industrial review, and technology assessment. Space mission design at the CDF complies with the 10-25 for the SEIM (Sec. 3.2.1).

The CDF is primarily utilized during the concept phase (pre-Phase A [5]). Fig. 4.1 lists the general CDF-level and Designer-level tasks to be completed for a mission or experiment.

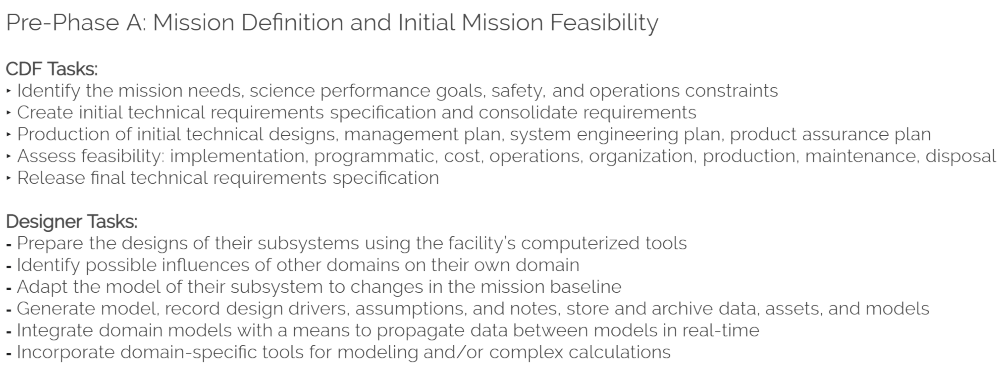

Figure 4.1: Pre-Phase A CDF and Designer Tasks. Extracted from [5].

4.1.1 CDF Stakeholders

CDFs contain at least one "main" room in which specialists gather. These representatives hold a "position" identified based on the subsystem they represent. CDFs have a variety of layouts and often have sub-rooms for conferencing, prototyping, and modeling. Fig. 4.2 shows a previous CDF arrangement at ESA/ESTEC layout and organization of space subsystem teams gathered for concurrent engineering activities [6]. Note that there are many additional stakeholders for a CDF, including clients from other industries, institutions, and universities. These additional stakeholders include project managers, ethics/law committees, military contractors, politicians, and international relations consultants. For COGENT, the CDF position representatives will take the primary focus, as including the previously mentioned external stakeholders is outside this project's scope. A high-level CDF Onion diagram was derived in the Appendix (Fig. C.1.)

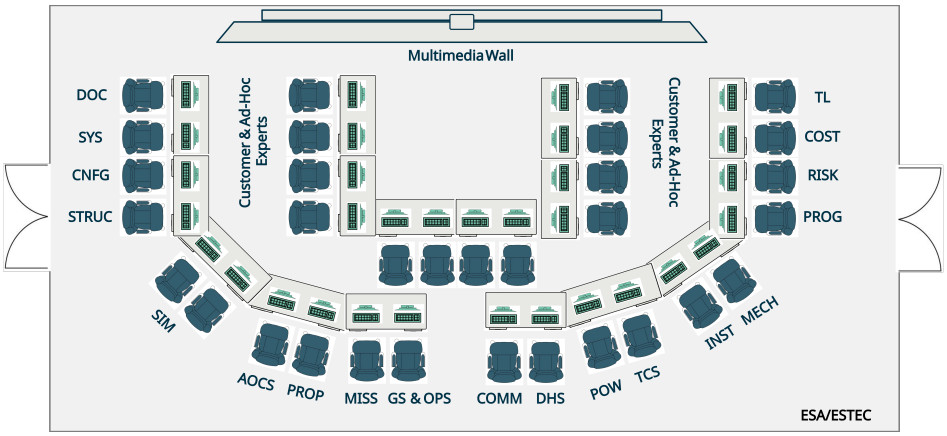

Figure 4.2: Example CDF Arrangement, based on the ESA/ESTEC Barracks [6]. For the full description of positions, please see Tab. B.1

4.1.2 Dimensionality Reduction

Fig. 4.2 contains 19 discrete positions and 9 (physical) subsystem teams in addition to customer representatives and ad-hoc experts. However, modeling and analyzing the concerns of a large number of stakeholders is challenging. The dimensionality of the challenge was constrained to three subsystem teams and one product management representative to reduce complexity. These subsystem positions include the Electric Power System (EPS)[1], the Attitude and Orbital Control System (AOCS), and the Propulsion System (PROP). A Product Manager model represents concerns at the Systems (SYS) position, and subsystem representatives include a simulation engineer, an architect, and a domain specialist. In order to verify and validate the FireSat case study, we gave these position representatives relative concerns and parameters (Table 4.1).

[1]Note that EPS and POW are different names for the same position.

Table 4.1: A small subset of CDF Positions, Representatives, Concerns, and Parameters.

Position ID	Representative	Concern	Parameter
SYS	Product Manager	Compliance	Cost
AOCS	Simulation Engineer	Usability	Accuracy
PROP	Architect	Maturity	Mass
EPS	Domain Expert	Synergy	Power Consumption

4.2 Customer Objectives Viewpoint

Keeping the aforementioned material from this section in mind, the Customer Objectives (C) viewpoint from the CAFCR framework identifies "*What* the *Customer* wants from the *Product* and *Why*." From HLR1 and HLR3, multiple designers must be able to score and rank architectures iteratively and concurrently. Before we can architect these processes, we must imagine how to conduct the process flow to derive a customer-centric product (Fig. C.2). The general process flow steps are to create initial models, score/rank models, merge multiple user score/rank results, tune models, filter/down-select, and repeat until there is team-based consensus in the preferred system architecture configuration.

Next, the process flow model is integrated with the base concurrent generative engineering domain model (Fig. 3.3) and visualized using MBSA components and relationships in the DAARIUS tooling (Fig. 4.3). In previous work, FoM could be either qualitative or quantitative [18]. For COGENT, we differentiate between quantitative, qualitative, and architectural attributes. Since automated architecture generation requires quantitative attributes, qualitative and architectural attributes require Human-in-the-Loop (HTIL) designers[2].

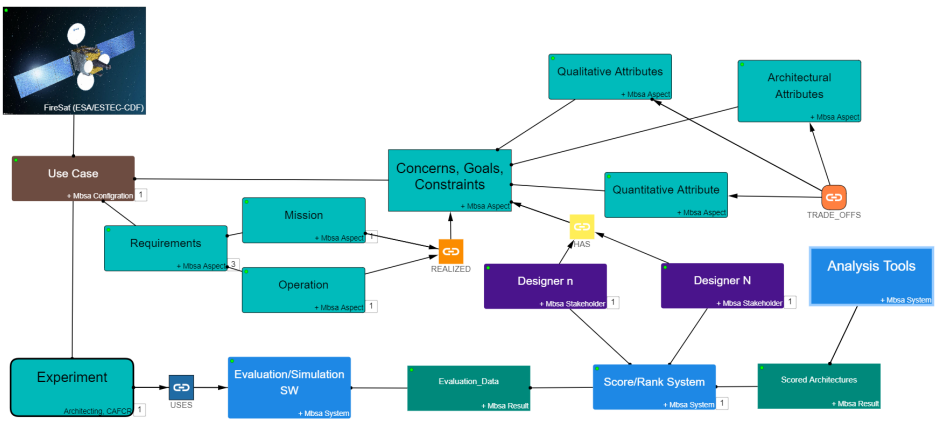

Figure 4.3: FireSat CDF Conceptual Model. An MBSA Model was constructed in DAARIUS. MBSA components colors represent: brown for Configurations, light green for Aspects, dark green for Results, purple for Stakeholders, and blue for Systems.

The conceptual model maps the mission and operational requirements as concerns, goals,

[2]Qualitative and architectural attributes are usually subjective whereas qualitative aspects are always objective.

and constraints. Below, four use cases (UC) capture customer objectives based on ESA's user requirements definition phase specifications [30]. These UCs are provided in narrative user scenarios and transformed into user stories. Each entry also briefly introduces the technological solution described within the UC.

4.2.1 Use Case 1 – Graph Database

UC1 is an implementation of HLR4, HLR5, and HLR6 using Graph Databases. Consider the following scenario:

"A system architect and data scientist consider using a graph database. The data scientist wants to understand the impact of changing subsystem components on other components. The data scientist views components and their relationships as equally important. The system architect wants to exploit the graph-like nature of architectures by storing them in a graphical format."

This narrative translates into two user stories:

US1: "As a data scientist, I want to use a Graph Database so that I can understand the impact of changing subsystem components, given that components and relationships are equally important."

US2: "As a system architect, I want to use a Graph Database so that I can store architectures in a graphical format, given the graph-like nature of architectures."

Graph Database Solutions

A Graph Database (GDB) is a database that consists of nodes, edges, and properties, to represent and store data in graph structures [31]. The graph relates the data items in the data store to nodes and edges, where the edges represent relationships between nodes. The relationships allow data in the store to be linked together directly and, in many cases, retrieved with a single operation (as compared to multiple JOINS in a SQL-like semantic query) [32]. Graph databases consider relationships between data as "first-class citizens" such that the relationship between different nodes is just as crucial as the node itself. Querying relationships is fast due to being perpetually stored in the database. For heavily interconnected data, visualizations for relationships are intuitive and intelligible using graph databases[3].

4.2.2 Use Case 2 – Experiment Tracking

UC2 is an implementation of HLR4, HLR5, and HLR6 using experiment tracking. Consider the following scenario:

"A design team uses experiment tracking. A product manager has a requirement that experiments must be verified using experiment tracking reports. A simulation engineer wants

[3]Fig. C.9 shows a Block Diagram for specifying the general structure of a GDB. GDB options are compared in Sec. C.4.2. Neo4J was the selected GDB solution for validation [32].

to compare architectures in two cases: (1) comparing multiple, discrete architecture configurations with varying parameters and (2) comparing a single architecture's performance indicators over multiple iterations."

This narrative translates into three user stories:

US3: "As a product manager, I want to use experiment tracking frameworks to verify experiments in reports, given that all metadata and parameters are tracked and stored."

US4: "As a simulation engineer, I want to compare multiple architecture configurations, given that discrete architecture configurations have varying parameters and components."

US5: "As a simulation engineer, I want to compare multiple "runs" of a single architecture configuration, given that varying parameters and components can influence performance indicators, design decisions, and ranking results."

Experiment Tracking Solutions

Experiment Tracking is a Machine Learning Operations (MLOps) process in which all experiment meta-information is stored in a single location [33]. Like qualitative scoring parameters, a single minor deviation in training data, training code, or hyperparameters can drastically alter a model's performance. Reproducing previous work requires the prior setup to match precisely. In the FireSat case study, experiment tracking is helpful in three ways: (1) tracking changes and comparing between various "runs" of the same architecture/model with varying parameters, (2) comparing different architecture configurations/models based on output parameters, and (3) comparing a single architecture with varying modes of usage (e.g. "solar mode" vs. "eclipse mode")[4]. As the concurrent generative engineering process is iterative, reliable and explainable tracking is required since model parameters, and user ranking/scoring values can easily change over time[5].

4.2.3 Use Case 3 – Data Storage

UC3 is an implementation of HLR1, HLR2, HLR4, HLR5, and HLR6 using cloud storage. Consider the following scenario:

"Users have individual workstations but need to combine their assets in a single location. A data engineer and a product manager have agreed that a cloud storage solution is preferred. The product manager requires a single source of truth for data, models, assets, and parameters for consistent solutions. A data engineer wants users to access integrated version control and continuous integration/development/testing. "

This narrative translates into two user stories:

US6: "As a product manager, I want to use a cloud storage solution for data, models, assets, and parameters, given that data objects are consistent in a single location."

[4] We decided that various modes of operation are effectively performance indicators and not separate configurations.

[5] Sec. C.4.3 contains a comparison of considered experiment tracking and management technologies. We selected Weights and Biases (wandb.ai) for the experiment tracking implementation [34].

US7: "As a data engineer, I want to use a cloud storage solution, given that some technologies have integrated version control and continuous integration/development/testing."

Cloud Data Storage

Having a single object storage location for multiple users is a non-trivial challenge (HLR1, HLR3). Additionally, the storage solution should have an unwavering read-after-write consistency such that there is no latency (delay) else data freshness between users is lost (HLR3). In recent years, cloud storage solutions have become ubiquitous for Agile Software Development and Information Technology Operations (DevOps) due to Continuous Integration, Continuous Development, and Continuous Testing (CI/CD/CT) capabilities which enhance software and data quality, scalability, reliability, manageability, and maintenance. CDF users must store a massive amount of structured, unstructured, and semi-structured data together with raw assets, documents, images, and mutating objects with an asynchronous push rate. For these reasons, a cloud object storage solution was selected instead of a polyglot of databases (e.g., a combination of relational, NoSQL, time-series, and graph databases)[6].

4.2.4 Use Case 4 – User-Defined Features

UC4 is an implementation of HLR4 and HLR5 using (custom) user-defined features. Consider the following scenario:

"A data scientist and a simulation engineer want to identify and analyze various relationships that exist within and between architectures. The simulation engineer wants to use custom artificial intelligence scripts to generate a correlation matrix of feature importance for designing experiments. The data scientist wants to use a natural language processing framework for keyword identification and sentiment analysis to discover patterns in user design decisions. These users have already created their custom scripts but would like to integrate them into the concurrent workflow."

This narrative translates into two user stories:

US8: "As a data scientist, I want to use user-defined features for natural language processing framework for keyword identification and sentiment analysis, given that there are patterns in user design decisions."

US9: "As a simulation engineer, I want to use custom artificial intelligence scripts to generate correlation matrices, given feature importance guides the design of experiments."

User-Defined Features and Domain-Specific Tooling

Since SCS natively supports Python, extending SCS to allow users to implement custom and personal scripts should be relatively straightforward. An integrated Python environment provides users with access to well-developed libraries[7] for machine learning, artificial in-

[6] Amazon AWS S3 buckets were selected for object/data storage and integrated with Amazon Glue Databrew ETL for some pre-processing and batch scheduling features. Other storage options are discussed in C.4.4.

[7] Example AI Python libraries include scikit-learn, TextBlob, and NLTK.

telligence, statistical analysis, and data visualization. Products that support user-defined features and custom Domain Specific Tools (DST) provide more "stickiness," keeping users within the application longer [35]. While DSTs allow for improved reusability and user-friendliness, any user tools developed must comply with the other requirements of the software and with interoperability in mind (via metamodels) [36]. Ultimately, Siemens will gain value if users can seamlessly import their Python scripts and programs into SCS.

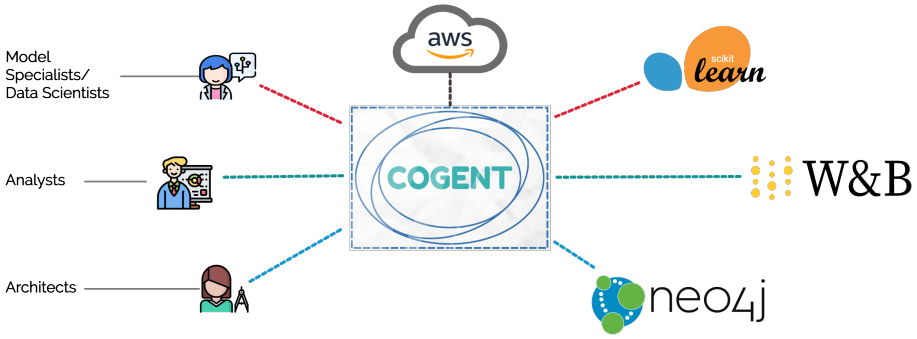

Figure 4.4: COGENT as a solution to connect users with technologies described in use cases.

With multiple users accessing multiple tools, we start to envision the most straightforward and most efficient approach for COGENT. Fig. 4.4 provides the starting point of using COGENT as a solution architecture to unite the various users with their UCs simultaneously. The next chapter will discuss how the user will use COGENT with the Application Viewpoint and the system/software requirements.

5 System Requirements

This chapter begins with the Application Viewpoint and COGENT positioning. Next, the core system/software functional requirements are introduced.

5.1 Application Viewpoint

From the CAFCR framework, the Application Viewpoint (A) details "*How* the *Customer* will use the *Product* and *Why*." This section first describes a system usage life-cycle scenario to define how the system is used concurrently with a Producer-Consumer relationship. Afterward, the context in which COGENT fits as a solution architecture is described.

5.1.1 System Usage Life-cycle

When developing a solution architecture, it is important to consider the life-cycle and context of the product. Users may misuse the product if the designer fails to consider a broad range of scenarios of usage (and misusage). The main system usage life-cycle scenario[1] includes the *Producer* as a design team generating architecture configuration and the *Consumer* as an analyst/management team. While not shown, the process is iterative, with the Producer team requiring multiple iterations of filtering, down-selecting, and annotating before involvement from the Consumer team. A deterring misusage of the system would be these teams working on the same model, but the model is not synchronized (two discrete instances or object locations). Three example system usage sub-scenarios are as follows:

Sub-Scenario A: A single design team (Producer) is exploring a potential experiment during Phase 0. A single analysis team (Consumer) is evaluating different architecture configuration concerning feasibility.

Sub-Scenario B: Multiple subsystem design teams (Producer) work concurrently to identify an optimal architecture configuration during pre-Phase A. One or more analysis/management teams (Consumer) are interpreting the results and prioritizing qualitative attributes via interactive dashboards.

Sub-Scenario C: Product Handover. A design team (Producer) transfers a product or domain knowledge to another design team (Consumer). The new design team becomes the Producer.

[1] Sec. C.3.1 includes additional system usage scenarios. Teams shows in Fig. 5.1 have two users but {2..*} users are allowed.

Based on these interactions, the aforementioned UCs, and the HLRs, we can construct the system usage scenario and draw the COGENT system boundary (Fig 5.1):

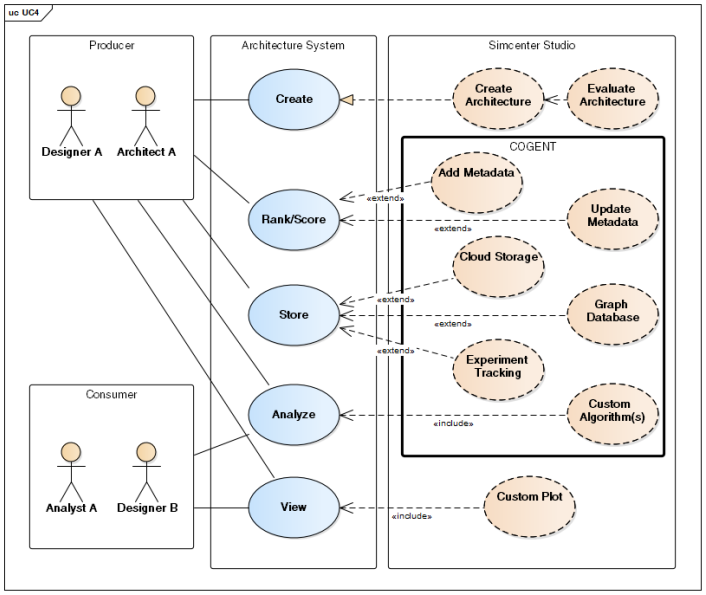

Figure 5.1: Use Case Diagram. Design teams are the Producer and analysis teams are the Consumer. The system boundary of COGENT is included.

5.1.2 COGENT Positioning

This section details how COGENT fits into the solution space. In the onion diagram shown in Fig. 5.2, the experiment (e.g. FireSat) is situated in the middle as a target viewpoint. COGENT behaves as a communication tool between concurrent design teams (technical design) and product/program management. Various teams are located in the CDF layer, while subsystems (e.g., EPS) are located at a subsystem layer. For roles, some examples (e.g., Simulation Engineer, Analyst) are located in the Role Layer. For a specific product, a Product Viewpoint can place an experiment at the center of the diagram (e.g., satellite, rover, launcher, etc.). The Technical Design side contains technical subsystems (hardware and software), and technical specialists (engineers). The Product Management side contains communication and analysis aspects of the project/product/experiment. Roles include managers and analysts (note that these persons are also technical, not to be confused as non-technical). Architects are situated between the Technical Design side and the Product Management side. Outside of the diagram are the tools that design teams (users) want to access.

In the appendix, Fig. C.1 acts as a high-level domain metamodel. Various experiments, subsystems (position), and users (representatives) are easily interchangeable. Fig. 5.2 is an applied version of this metamodel, tuned for the FireSat case study.

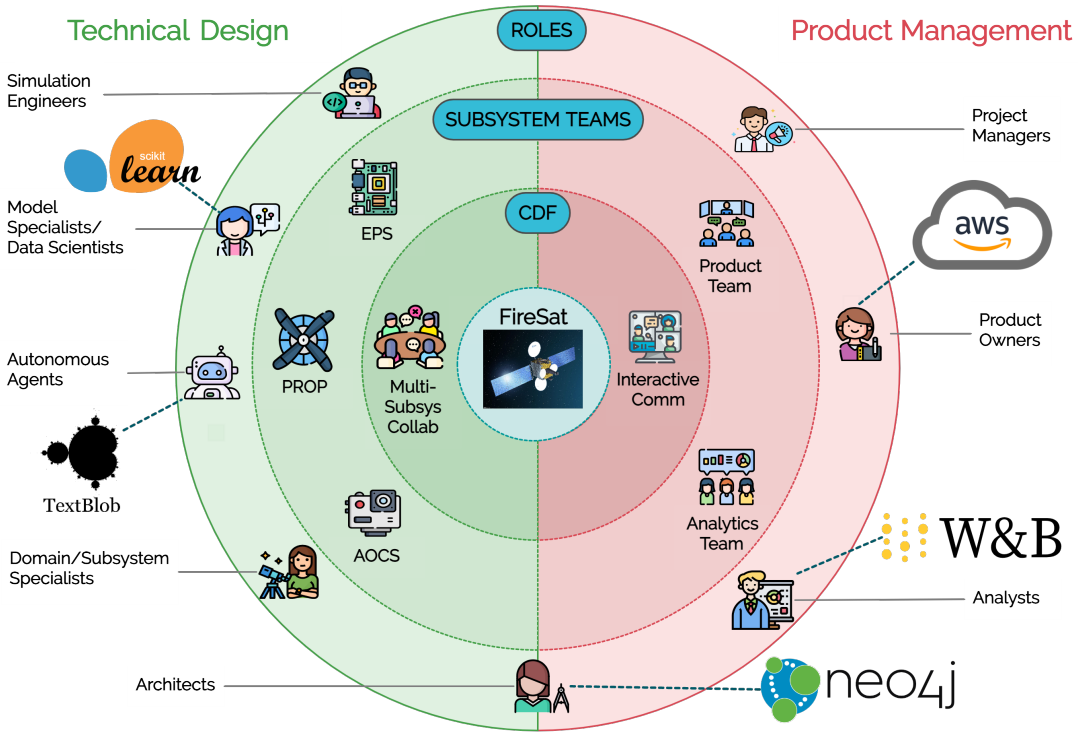

Figure 5.2: FireSat Onion diagram, updated from Fig. C.1. Specific subsystem teams have been added along with external tools the users wish to use.

5.2 Functional Requirements

This section contains the derived primary functional requirements for software development. For conciseness, not all requirements are listed here, and the labels were pruned (please see Sec. B.2 for a full list of requirements). Each requirement contains an Identification Number (ID), a Priority (P), and a brief Description. The Siemens SISW specification for Priority was adopted with ranges from *High* (H), *Medium* (M), and *Low* (L). A Priority value of *H* is essentially a "must," a value of *M* is approximately a "should", and a value of *L* is effectively a "could/won't" (MoSCoW prioritization format). Additionally, the priority values of *Transferred* (T) means this requirement was transferred to the conceptual team at Siemens SISW. The requirement structure is based on the software requirements phase definition for space systems at ESA [37]. The Priority (P) column is also tagged with a color; green indicates the requirement was fulfilled for verification, and yellow means the requirement was partially fulfilled.

Table 5.1: Primary Functional Requirements.

ID	P	Description
R1	**H**	The system shall support N concurrent user instances.
R1.1	**H**	The system shall integrate the scoring results from multiple users.
R1.2	**T**	The system shall rank the combined results from user scoring.
R2	**H**	The system shall plot all architecture configuration FoMs with Pareto charts.
R3	**H**	The system shall allow for users to interact with charts and diagrams.
R4	**H**	The system shall allow users to input their design decisions.
R6	**H**	The system shall allow for user-defined algorithms, features, and parameters.
R7	M	The system shall differentiate between quantitative, qualitative, and architectural FoMs.
R7.1	M	The system shall automatically generate scores for architecture configurations based on quantitative, qualitative, and architectural FoMs.
R7.2	M	The system shall automatically rank architecture configurations based on quantitative, qualitative, and architectural FoMs scores.
R7.7	M	The system shall automatically combine the scores from quantitative, qualitative, and architectural FoMs.
R15.1	**H**	The system shall utilize user-inputted "likes."
R17	**H**	The system shall store all relevant (meta)data.
R17.1	**H**	The system shall support storing metadata at the appropriate level (sys/comp).
R17.2	**H**	The system shall support storage into a Graph Database.
R17.5	**H**	The system shall support storage into a Machine Learning Experiment Tracking datastore.

6 System Design & Architecture

This chapter first describes the COGENT domain model followed by a distinguishment between monolithic and modular software architecture frameworks. This is continued with description and comparison of two architectural patterns. Afterward, the Functional viewpoint (F) from the CAFCR model is addressed for the COGENT Plugin System description, followed by the Conceptual and Realization viewpoints (CAFCR). Since model-to-model transformations became a large part of the project, we consulted the Model-Driven Architecture (MDA) standard. Additionally, since Python was the primary programming language, the Class diagrams have been adapted to match the Python implementation methods. For instance, functions equate to static methods, data types match the Python description (e.g., "None" instead of "void"), and data types include the data structure (e.g., "dict(str)" instead of "str"). The MDA standard allows for this type of flexibility in modeling.

6.1 COGENT Domain Model

This section contains the COGENT Domain Model (Fig. 6.1. The model is an extension of the Base Concurrent Generative Engineering Domain Model (Fig. 3.3). In this version (Variant 2), COGENT exists within the GE Module (within SCS). COGENT contains conceptual services like *(interactive) dashboards*, *scoring matrices*, and *FoM weights*. Additionally, the COGENT is where the *Plugin Architecture* resides. For an alternative design in which the COGENT is external from SCS, please see Fig. C.4. This decision was made together with Siemens SISW as end-users should stay within SCS when selecting architecture configurations.

Fig. 6.6 shows a high-level physical view of a process starting from an ACEL Model to storing FoM scores, dashboard visualization, and storing of metadata (scores). Various data formats and data types have transitions within the high-level activity.

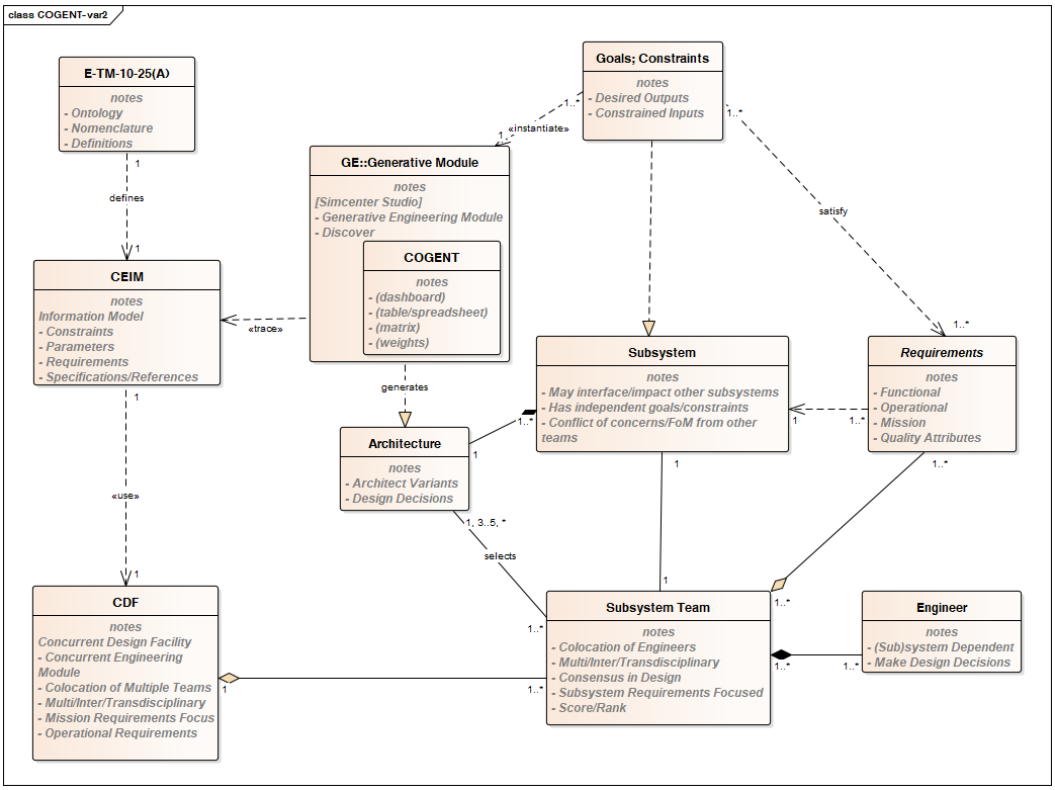

Figure 6.1: COGENT Design Alternative (Variant 2). COGENT is embedded in SCS.

6.1.1 Monolithic Versus Modular

Before approaching the requirements, it is critical to consider the environment in which the software will run. Software engineering typically considers whether an application should have either a monolithic or a modular framework. Monolith frameworks have tightly coupled behaviors, where the software natively supports all of the considered technologies required for use cases. A modular framework is loosely coupled and includes only minimal functionality and structure, requiring a plugin or service to enable additional functionalities. In software architecture, tightly coupled generally means that software modules are highly cohesive (dependencies) and make assumptions regarding all other software modules. Loosely-coupled software modules have low cohesion, and independent modules communicate via standardized, neutral interfaces.

Siemens SISW clients have significant variations in use cases. Other than at high levels of abstraction, use cases between the automotive, aerospace, and medical domains are not similar or comparable. Even when only considering the aerospace domain, two different products/projects will require very different technologies for development and analysis. As monolithic frameworks focus on implementing commonly used features, they are not suitable for our solution architecture. Modular architecture patterns support strong encapsulation for hiding implementation details, well-defined interfaces for things that are not hideable, and explicit dependencies when describing and expressing co-module relationships. We decided

to implement a modular solution architecture since nearly all use cases are supported, and the long-term maintainability is less expensive.

The following two subsections introduce the Microservices Architecture and Plugin Architecture patterns.

6.1.2 Microservice Architecture

The Microservice Architecture pattern provides the opposite behavior of the monolithic pattern in that services are small, independently deployable, decentralized, and meant to be autonomous [38]. Autonomous here means that developers are responsible for making their own decisions when creating their services. Microservice architectures are primarily useful in deploying API REST-based topologies in websites and browsers using HTTP. A significant benefit is that this is a distributed architecture, such that all components are fully decoupled and accessed through remote access protocols (e.g., REST, SOAP) [39]. Some limitations for microservices are that modules are intended to be independent of other modules, add complexity, and require interfacing and communication with the core software platform [40]. This interfacing with the platform or other services can become complex or complicated while also carrying a risk that the developed microservices can become deadlocks if aspects of the core software platform (SCS) change.

6.1.3 Plugin Architecture

The Plugin Architecture pattern, formally known as the Microkernel Architecture pattern, sits between the monolithic and microservice architectural patterns [39]. The Plugin Architecture pattern consists of a centralized monolithic core and microservice-style plugins. Plugins generally do not know about other plugins, though they can interact via interprocess communications with the core. Plugins support abstraction, minimize complexity, and force software components to be modular [41]. This pattern does not specify implementation details, but rather that plugin modules are required to remain independent from each other [42]. This architectural pattern is ideal whenever a base software platform (SCS) needs to be extended or customized for clients.

6.2 Functional Viewpoint

This section initializes the Functional Viewpoint (F) from the CAFCR methodology, specifying the feature specifications of the solution architecture. The functional viewpoint is meant to answer the question "*what* is the final product?" First, a domain model explores the entire SoS and identifies where our System-of-Interest (SoI) fits in as a solution architecture.

6.2.1 COGENT Plugin System

Based on the analysis in Sec. C.4.1, we selected a plugin-based system. COGENT derives from a popular PyTest Plugin Manager called Pluggy [43]. In our plugin architecture, a

Plugin Manager acts as a core program that utilizes *hooks* to construct/destruct plugins. The *hook specifications* (hookspecs) are defined in the Plugin Manager describing how a plugin can be registered. At the Plugin level, a *hook implementation* (hookimpl) is defined and describes what the plugin will do. The Plugin Interface is responsible for communicating these hookimpls to the hookspec.

Data Format Transformations & Communications

We can see some requirements when evaluating the selected technology data format requirements with the provided evaluation data. Firstly, data models will undergo several transformations to be compatible with the deployed tooling service. We selected the JSON format since it is the de facto standard for unstructured data formatting. Additionally, most of the selected technologies directly support JSON in some way. As such, all architecture configuration input data (e.g., evaluation data) was converted from CSV to our centralized JSON format to help mitigate issues during development and provide a consistent, extendable/appendable data format. Secondly, the communication over APIs or other interfaces will be separate from the data transformation to encourage reusability since the same JSON model is appropriate for different plugins. The goal is to prevent data duplication, additional transformations, and network access for reading/writing.

Design Patterns

During the initial design, three types of plugins we considered. The Adapter Design Pattern is utilized for data conversion/transformation (e.g., converting CSV to JSON) [44]. A Polyglot Design Pattern specifies how to manage multiple data assets moving through the pipeline to different data storage/database solutions (e.g., GDB, experiment tracking, AWS) [45]. The Strategy Design Pattern is for custom algorithms (e.g., a weighted ranking/scoring algorithm using simple statistical methods) [44]. Fig. 6.2 shows the Pattern Classes in a Block Definition Diagram. Functionally, COGENT acts as data transformation entities since input data is transformed and used to produce new data, transfer data to storage solutions, or used in plugins.

Plugin Ordering

Next, we needed to consider the timing of plugin activation since out-of-order executions can lead to deadlocks, broken pipelines, or inconsistent data/objects. COGENT supports timing flags, including the self-evident labels of "try first" and "try last." One additional ordering label is called a "hook wrapper," which is triggered when a plugin module initially executes and fires before any other timer activation (i.e., before "try first"). The hook wrapper indicates the calling of a specified function to wrap all other standard hook implementation calls. Besides these order generator labels, the COGENT system inherited a "Last In, First Out" scheduling scheme from Pluggy.

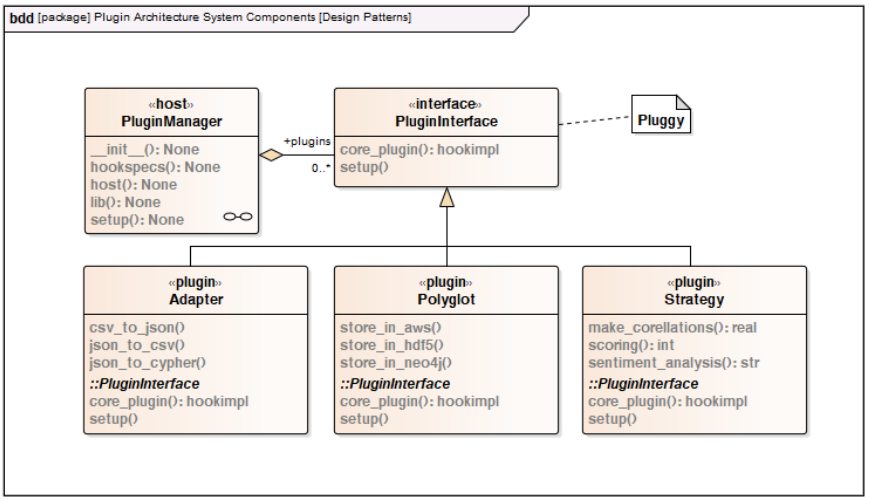

Figure 6.2: COGENT Plugin Architecture with Design Pattern Classes for implemented Plugins.

6.2.2 COGENT-Specific Functional Requirements

With the specification of the COGENT plugin architecture, the plugin-based functional requirements were updated. The following specifications match the description from Sec. 5.2 and provide specific requirements for ensuring the system is developed correctly. The previously defined non-functional requirements also apply to COGENT as a solution architecture.

Table 6.1: COGENT-Specific Functional Requirements.

ID	P	Description
R21	H	The system shall support plugins.
R21.1	H	The system shall support N concurrent plugins.
R21.2	H	The system shall support the ordered timing of plugins.
R21.3	H	The system shall support custom user plugins.
R21.4	H	The system shall support deregistering of plugins.

Next, the functional decomposition of COGENT is addressed.

6.2.3 COGENT Functional Decomposition

Previously, we defined the SoI boundary and the system requirements. Next, we derived a general systems functionality description model visualizing the functional placement of COGENT[1]. The Systems Functionality Description showcases the behavior of the high-level task workflow, the functional decomposition of systems, and expresses system capabilities.

[1]The SV-4 Systems Functionality Description and SV-10c Resource Event-Trace Description models are from DoDAF 2.02.

Fig. 6.3 below illustrates an overview of the entire SoS. The green, dashed rectangle demonstrates our COGENT solution architecture as the SoI interfacing with SCS. SCS has both a Notebook and Discover as visual User Interfaces (UI). Some of the tasks are abstracted away to avoid additional clutter in the figure. These activities include sending merged models to Discover, interfacing each subsystem decomposition with the Cloud Data Store, and the HITL tasks of visualizing GDB/Experiment Tracking results. Other HITL actions like "discussing trade-offs" (collaborative) are also not shown as activities since these activities are not responsibilities of the system.

On the other hand, HITL actions like generating quality attributes, scores, likes, and design decisions are shown since these will be input (and guided) by the COGENT system. Here, "generate" means the user is creating these knowledge objects. Note that the Scoring/Liking/Ranking activities are completed in Discover, but the front-end aspects were emulated because the functionality had not yet been implemented at this point in the project.

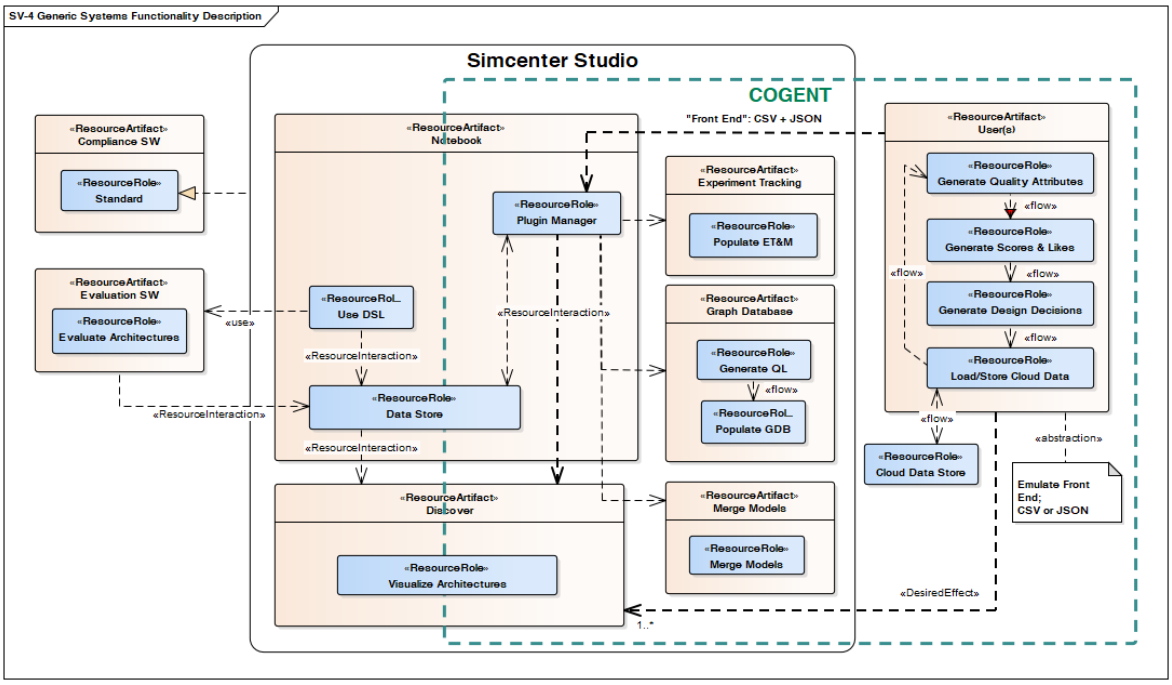

Figure 6.3: General Systems Functionality Description. The model demonstrates the high-level flow behavior of the system.

Interface Layers & Functional Blocks

For a better understanding of the functions and interfaces in our SoI, we decompose the entire system into layers and functional blocks. Fig. 6.4 demonstrates the layers and SoS interfaces for the COGENT implementation with plugins, data format transformers, and connections to other systems through SCS (i.e., Siemens-based interfaces). As such, it is possible to route information from Siemens Amesim or NX to a specific plugin.

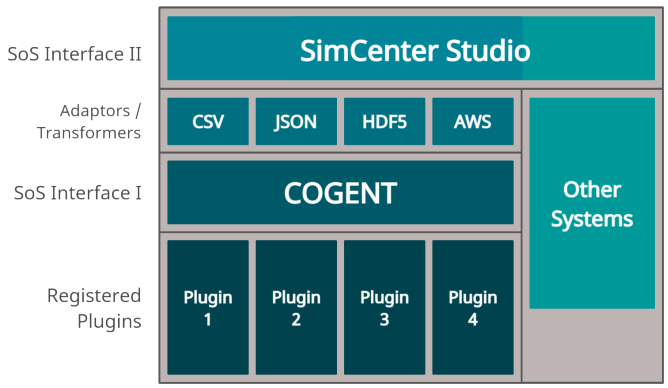

Figure 6.4: COGENT Plugin Architecture Interface Layers and Functional Blocks.

Defining User Classes

Now, we introduce the concepts of an *Orchestrator* user and a general *User*. The *Orchestrator* is, in general, the lead system architect/engineer that is guiding the concurrent generative engineering activity at a CDF. The *User* is a general subsystem representative that is following the *Orchestrator*'s guidance. This distinction is essential as CDF activities require organization, and at least one lead representative must be responsible for the orchestration. The *Orchestrator* selects which system architecture configuration(s) to discuss, score, rank, and select for further mission phases.

Next, a resource event-trace description diagram demonstrates the (general) critical-path sequence of events[2]. In Fig. 6.5, "Staging" is a Software Development and & Operations (DevOps) term meaning the data has been loaded into the SCS environment. The *Host* lifeline represents a process *Orchestrator* engaging the Plugin Manager's automatic tasks (assuming these plugins are registered). During the Rank/Score lifeline, a single user instance can analyze the models they have pulled (downloaded). The Host merges and transforms the models into other formats (query languages or parameter vector form) for usage in third-party or user-defined features. Finally, all objects and assets are stored in the specified centralized database/object storage location.

[2]Here, critical-path means the path in which all categories of plugins are loaded and each process is conducted exactly once (no iterations).

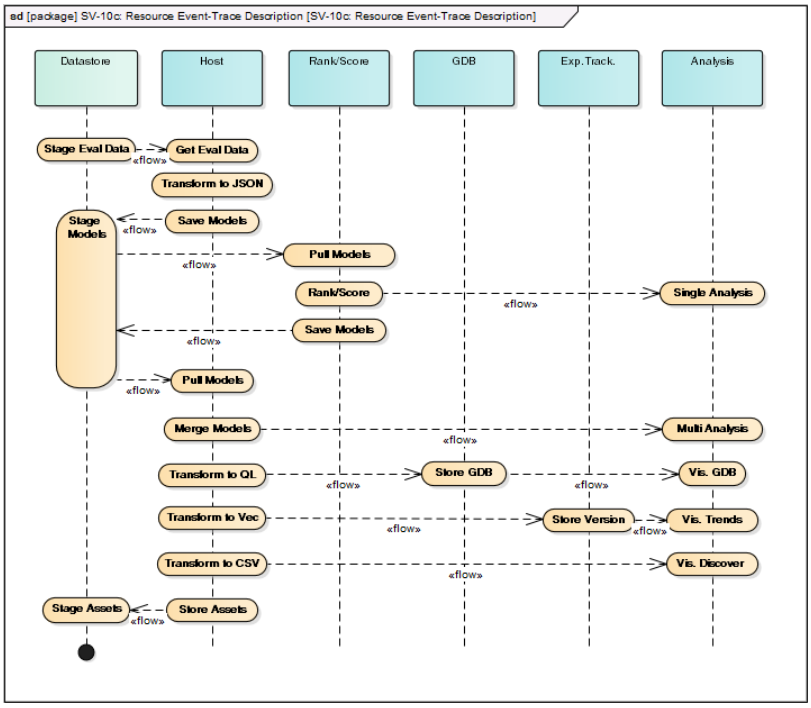

Figure 6.5: Resource Event-Trace Description Diagram. This diagram illustrates the general flow of tasks in a sequential-like behavior. This model assumes that plugins for GDBs, Experiment Tracking, and User-Defined Features have been registered.

6.3 Conceptual Viewpoint

The next section will dive into the Conceptual Viewpoint from CAFCR, making the solution architecture more specific. Before we develop activity diagrams or sequence diagrams with specific technologies, the evaluation data must be described.

6.3.1 FireSat Evaluation Dataset

For the FireSat case study, a dataset of 64 architectures was generated using ACEL and quantitatively evaluated with Siemens Amesim. Multiple EPS and AOCS subsystem configurations contain various batteries, solar panels, fuel types, propulsion components, and actuators. The ACEL model specifies the ownership, multiplicity, component coupling, connection constraints (ports), and interoperability requirements are architectural attributes. For example, a configuration can have either a magnetorquer or an electric propulsion system. The EPS owns batteries and solar panels (ownership). Some propulsion systems require a specific fuel type (tight coupling). A solar panel can have either one or more batteries in a series, parallel, or series/parallel configuration (multiplicity and interoperability). Additionally, the component-level quantitative values are also specified in the ACEL files (e.g., nickel-cadmium battery power consumption and mass).

ACEL initially generates all architecture configurations. Next, Amesim runs a computational simulation to calculate the system-level quantitative values[3] (e.g., total consumption and total mass). The system-level quantitative attributes are used for system architecture configuration comparison and filtering in SCS Discover. Previously it was imagined that the *Orchestrator* and *Users* primarily down-select architectures configurations based on the system-level performance and behavior. With COGENT, the *Orchestrator* and *Users* analyze down-selected architecture configurations together or as individuals with additional tooling or user-defined features.

Amesim stored the 64 initial architectures in a tabular CSV file. Note that all subsystem components (e.g., batteries, panels) are stored in a single CSV cell as a concatenated string of components. Fig. 6.6 provides the conceptual high-level activity diagram integrating these components with the new scoring/ranking activities and considers users inputting their quantitative/qualitative attribute (FoM) scores[4]. For the COGENT prototype, qualitative scores are *Integers* between 1 and 5 (very poor to very good). Quantitative values are from the FireSat Dataset are *Real* numbers. Design Decisions are inserted as *Strings*.

The next step was to transform the CSV file into a JSON format and separate the subsystem components based on their type (e.g., battery, actuator). This initial model transformation is described in the next section.

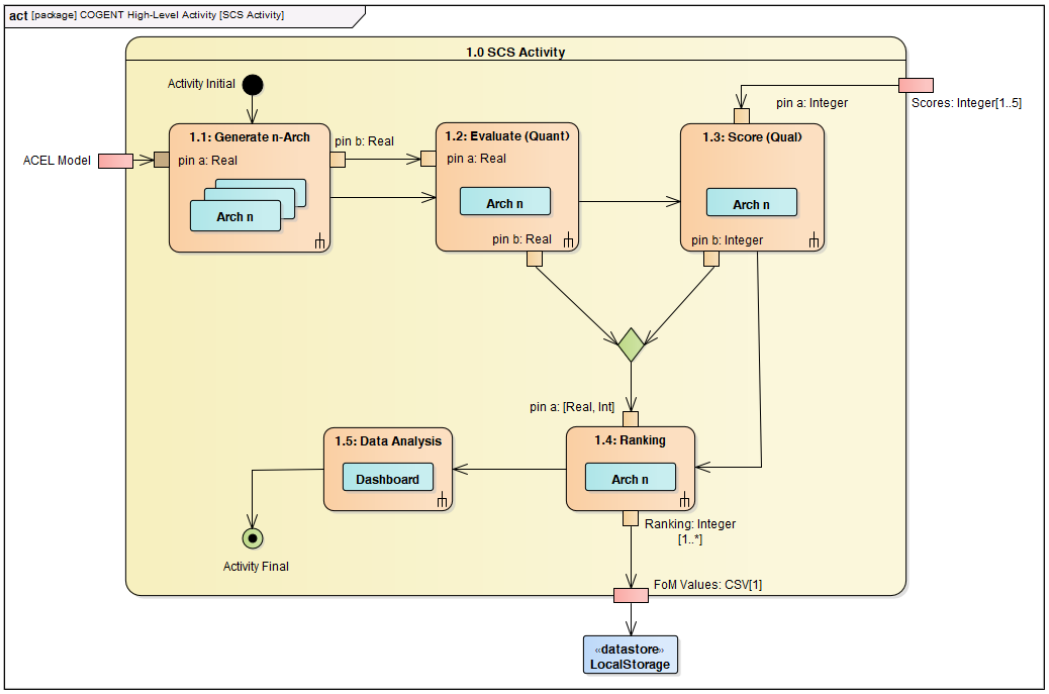

Figure 6.6: COGENT High-Level Activity Diagram describing input/output data types and (conceptual) parameter value ranges. The results of quantitative and qualitative FoM evaluation and scoring results are merged and made available for a Ranking Activity.

[3] ACEL maps attributes to parameters as architectural attributes.

[4] Siemens SISW uses the term Figure of Merit (FoM) for all attributes, including Key Performance Indicators (KPI).

6.3.2 Initial Model Transformation for COGENT

A significant limitation of the CSV file format is that tabular data can only be structured. Semi-structured/unstructured data cannot be used; thus another file format is needed for NoSQL databases (including GDBs). As previously stated, JSON is preferred as the baseline data format. The first step is to convert all CSV data into JSON format. Since JSON is semi-structured, it is also possible to store metadata in layers, i.e., at the subsystem or component levels. Recall that a battery has its own parameter values and architectural attributes. We decided to organize the semi-structured data format such that component or subsystem parameters and user design decision notes can be stored with relationships at the component or subsystem level.

Relationships impact many aspects of the design space exploration, as changing a single component influences other system components. As a result, relationships highly influence design decisions based on qualitative, quantitative, and architectural attributes. The semi-structured model can store subsystem and component level design decisions directly with the parameters and components that influenced that design decision. Initially, a metamodel was developed to describe internal relationships [46] and a normalized logical model for both column-based Relational Database Management Systems (RDBMS) and NoSQL databases [7] was constructed (schema) (Fig. C.10). The schema is useful as a metamodel for mapping relationships, constraints, and parameters to other models for implementation. Additionally, architecture configurations were assigned IDs to match the schema (e.g., ARCH001), and components were given a class, e.g., "Pulsed Plasma Thruster" is a member of "Propulsion."

Global & Local Configurations

Next, a distinction between the global/orchestrated configuration class (Config) and the user-specific configuration class (User) is made. The *Orchestrator* will update the global configuration file specifying the specific architecture configuration(s) to score and which databases to use. This global configuration will be sent to users and integrated with their user-specific configuration file containing parameters and paths when modules or plugins are executed (e.g., User ID and Local Model Path parameters). The initial model transformer inherits the global configuration parameters to ensure *Users* cannot use the wrong database or operate on the wrong architecture configuration. The *Users* are not authorized to edit the global configuration file, as only the *Orchestrator* is permitted to change critical parameters.

Fig. 6.7 displays a Class diagram for the initial data transformation, the configuration file inheritance, and the centralized AWS S3 storage classes for the *Orchestrator*. An S3Upload class inherits the database and experiment name parameters from a Config class. The InitDataTransformer class inherits an architecture ID and evaluation dataset path from the Config class to transform initial CSV files into JSON architecture model files. The S3Upload class then inherits the S3 access key parameters from a UserConfig file based on a specific user (the *Orchestrator* in this case). Amazon AWS S3 inherits the JSON object through an AWS interface configured in S3Upload.

The next task is for *Users* to rank and score the configurations, along with analysis with user-specific features. However, we first need to describe the decision to keep data storage access independent from the plugin architecture.

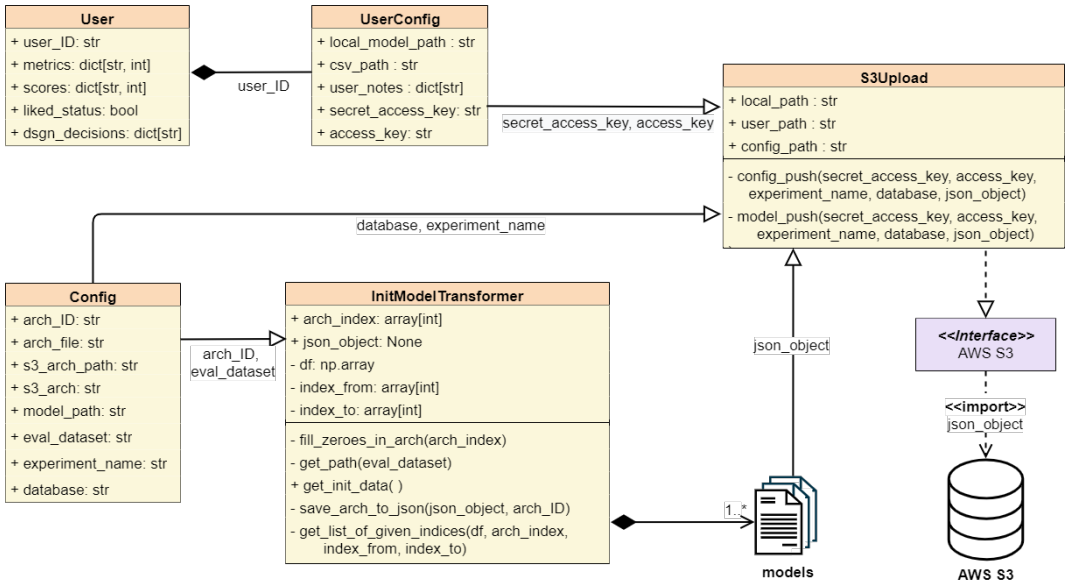

Figure 6.7: Class Diagram for the Initial Model Transformation Class with the configuration file inheritance and the centralized AWS S3 storage Classes for the *Orchestrator*.

6.3.3 Cross-Cutting Concerns: Data Storage

Data concurrency, consistency, and read/write access conflicts in multi-user systems are common challenges when interacting with a single file or asset. For example, if two designers want to save their work simultaneously, they might have access conflicts. A typical example of data inconsistencies is when a developer reads specific data, and another designer changes (writes) to the same datastore after the first designer has acquired the data. The "freshness" of the data is lost, and the data has inconsistencies.

Aspect-Oriented Programming (AOP) introduces the concept of cross-cutting concerns as well as the risk of "tangling" or "scattering" (meta)data during storage [47]. Cross-cutting concerns are aspects that cannot be completely decomposed from the system. Tangling is when a single module interleaves or has inter-dependencies with multiple requirements, whereas scattering is when a single requirement has multiple design or code modules [48]. For this reason, a trade-off has to be made between the system's autonomy and the user's responsibilities related to the initial model transformations and data storage. If integrated into the plugin manager, tangling and scattering concerns would be problematic for our initial transformation and data storage access behavior. Please consider the following two cases:

Data Storage Scattering

First, the plugin manager has the behavior that it will execute all registered plugins every time it is called. If the initial model transformation module is registered, it will continuously convert the CSV evaluation dataset into JSON models (non-ideal). Data duplication or inconsistent data (since the models can be overwritten) may occur. However, we do not want to forbid

this activity completely since a change in the evaluation dataset (new parameters) would be desired for our models. As such, the initial transformation activity is separated from the plugin manager.

The second case is similar for storing JSON objects and assets in a centralized location. Data may be overwritten, corrupted, or duplicated if the plugin manager behavior encapsulates the reading or writing aspects. Additionally, the network traffic and object access instances will increase significantly and without merit while using cloud storage solutions—this traffic leads to an unwanted accumulation of cloud-service costs and resources. Again, we do not want to forbid the behavior as there may be cases in which a user needs to redownload an architecture configuration (updated parameters) or reupload an architecture configuration (change in score metadata or additional design decision notes).

Based on these two cases, we decided to de-couple the initial model transformation process and all data storage access processes from the plugin architecture. Next, we will visualize our conceptual process flow.

6.3.4 Conceptual Process Flow

After multiple users score and rank the architectures, the results are merged in a combined architecture JSON model. Optionally, analysis can occur at this step. Afterward, the combined architecture model is used to generate necessary query languages for GDBs or into parameter vector form for experiment tracking (depending on registered plugins). Finally, all relevant objects and assets are stored in the centralized data storage container or warehouse. Fig. 6.8 provides a visualization of these steps in their respective layers with a sequenced timeline including an iteration loop. Example technologies are provided under the Storage and Plugins layers.

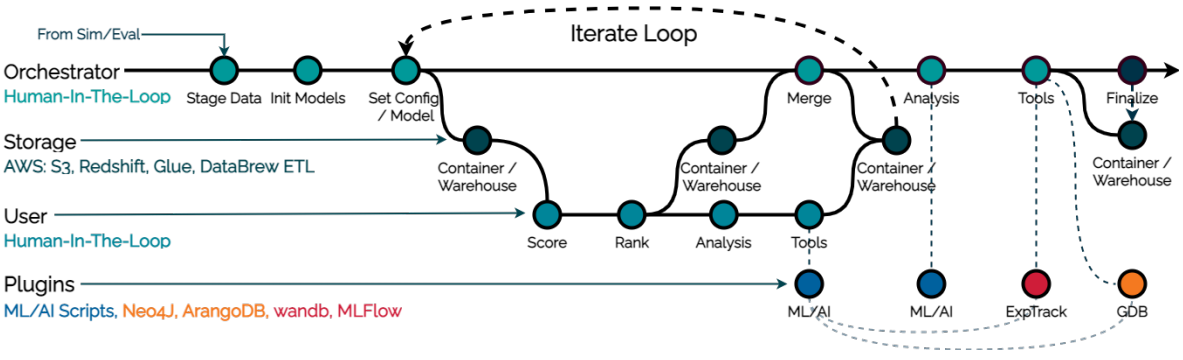

Figure 6.8: Process Flow Timeline with an Orchestrator, Storage, User, and Plugins layer. The Storage and Plugins layers include example technologies that can be integrated into the system.

6.3.5 Multi-User Model Merging

Initially, we considered two different strategies for storing user metadata related to FoM scores, rankings, and likes. The first method was an orchestrated integration and the second method was a concurrent integration.

Orchestrated Merging

The first strategy was for each *User* to download the selected architecture model, complete a task while appending the new metadata to the model, and then upload the model back to the centralized storage. In this way, the baseline model would be updated after any user task. In this way, the model is always merged and/or updated in near real-time. However, users cannot complete HITL tasks, like scoring and adding design decisions, in near real-time as they need to think and reflect on single architecture configurations attributes and cross-model comparisons. Consider the case: *User A* downloads a model and begins a task. *User B* begins and completes a task before *User A* completes their task. *User A* and *User B*'s models now have conflicting or missing information when merging to the same baseline model. An additional HITL task is then necessary for handling merge conflicts (Fig. C.5 illustrates this process in Appendix 2).

Concurrent Merging

To avoid inconsistencies, we implemented a different strategy. Each *User* downloads a baseline architecture model containing the architectural components and parameter evaluation values, which is used in the architectural visualization. Whenever the *User* completes HTIL tasks, a new version of the model is created and tagged with the *User*'s ID. The *User* then stores their discrete model in the centralized storage area. The *Orchestrator* sets a deadline for the completion of all HITL tasks during the current design sprint. Then, the *Orchestrator* will merge the baseline model with all *User*-created models, appending the metadata to a merged model. If a *User* does not complete an HTIL task, then this task's contribution is omitted. Fig. 6.9 illustrates the conceptual activity diagram for the subjective parameters score merging activity (qualitative and architectural attributes).

The next section will describe the realized technologies and provide examples of their implementation results.

6.4 Realization Viewpoint

The Realization Viewpoint (R) from the CAFCR methodology describes the results of the actual technologies (plugins, tools) and the actual implementation. This section briefly demonstrates the results of connecting COGENT in SCS to third-party tools and user-defined scripts for the FireSat case study. Next, an overview of the system components is described.

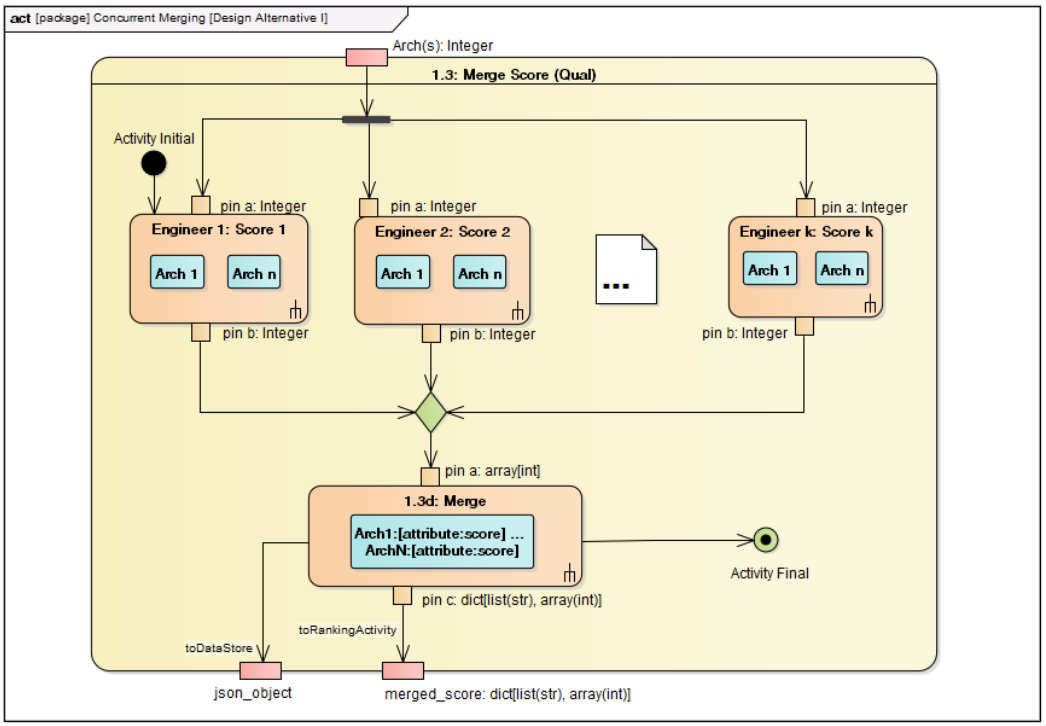

Figure 6.9: COGENT Conceptual Concurrent Score Merging Activity Diagram. The qualitative/architectural attribute scores from multiple Users are merged as an Orchestrator's task.

6.4.1 System Components

After finishing the description of all core features for the plugin manager and storage modules, specific plugin implementation details are relevant. Fig. 6.10 illustrates a functional system component diagram that is color encoded to also show the functionalities for each block and component. Both the local (on device) and AWS S3 cloud object storage technologies are shown. The initial firesat.csv evaluation dataset is converted into JSON models and fed through the pipeline. Registered plugins include Neo4J for the GDB, wandb for experiment tracking, and two examples of user-defined artificial intelligence Python modules (scripts) using libraries including: scikit-learn, matplotlib, seaborn, NLTK, and TextBlob.

6.4.2 Developing Plugins

As previously described in Sec. 6.1.3, plugins interact with the COGENT plugin manager. In development, all plugins begin as Python modules with a single task. Two primary tasks must be completed for this to use the plugin: discoverability and setup file definition. The first task augmenting the Python modules to be registerable with the plugin manager (hook implementation) and augmenting the plugin manager's hook specification file must have knowledge of the specific plugin.

The second task is to create a setup file to easily build the plugin with all necessary li-

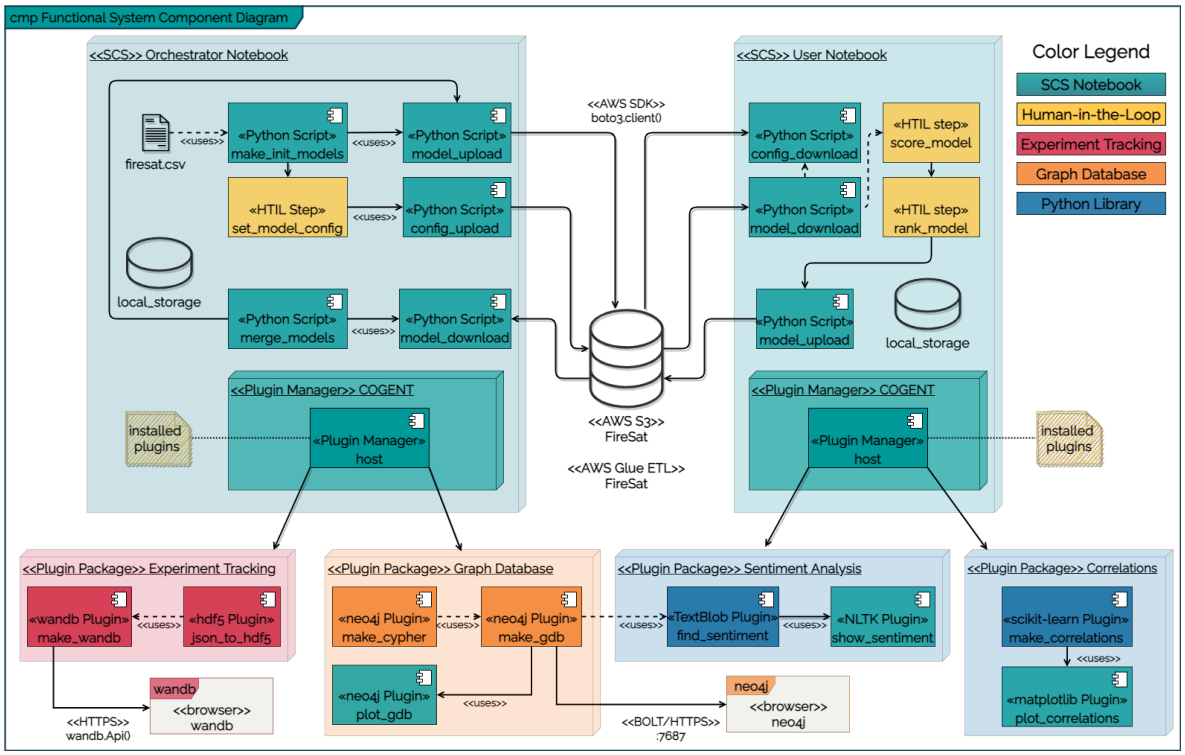

Figure 6.10: Functional System Component Diagram with color encoding for Functionalities and HITL steps. The left SCS module is the *Orchestrator's* Notebook responsible for the initial model transformation, storage access, and merging. The right SCS module is the *User's* Notebook for scoring and ranking, along with storage access. The bottom four blocks are implemented plugins for Neo4J, wandb, and two user-defined scripts.

braries/dependencies and the entry point for the plugin interface to discover. An entry point is a module (Python program) location where a function or method will trigger the program to run. The entry point trigger is usually placed where the Class is defined (though this is not required, and a single module can have more than one entry point trigger if desired). The entry point is also where the plugin ordering is specified (e.g., try first).

6.4.3 Autonomous Agents

To test that a multi-user system is working correctly as a single developer can be challenging if only using manual testing procedures. Additionally, gaining access to designers working in a CDF to evaluate the system is a challenge of its own. Instead, autonomous *Agents* were created to emulate the design tasks for CDF representatives. Autonomous *Agent* tasks include selecting quality/architectural attributes, scoring attributes, liking or disliking an architecture configuration, and writing design decisions. At the time of this project, there was no UI for users to add metadata, parameter values, or design decision strings. However, we assumed that the interaction between the User and SCS would generate and store metadata in JSON format. Effectively, the *Agents* need to complete tasks that would generate equivalent

metadata, parameter values, and design decision notes and store this meta-information in a JSON format.

All 64 architecture configurations were tested with three simultaneous *Agents*. A smaller subset was tested with five simultaneous *Agents*, but there was no noticeable impact on the quality of the emulated process (only performance and resource access costs). *Agents* were representatives of the EPS, AOCS, and PROP subsystems. First, a dictionary was created to include various quality/architectural attributes, and the *Agent* could randomly select three. For consistency, the first *Agent's* three attributes are used by the other two *Agents*. For this report, the attributes of "Usability," "Maturity," and "Synergy" were selected. Usability is defined in IEEE Std. 610.12 as enabling the user to meet their specified goals with effectiveness, efficiency, and satisfaction. Maturity is a risk-based qualitative proxy for the technological maturity status of a mission, or an instrument [18] (similar to NASA's Technology Readiness Level [49]). Synergy is a qualitative (architectural) measure regarding complementarity asset capabilities, such that overlapping their capabilities exist, then the asset with a lower relevance causes its architecture configuration to be non-synergistic [18].

Autonomous Agent Tasks

Based on the three selected attributes, the *Agents* would randomly score each attribute with a value between one and five, with one being "very poor" and five being "very good." This means a "Usability" score of "5" is considered "very good." Next, a simple "liking" algorithm was created such that if the cumulative score of three attribute values was greater than 12 (80%), then the *Agent* "liked" the architecture configuration. Otherwise, the *Agent* "disliked" the architecture configuration. The liked status was represented as either a 1 for like or -1 for disliked (and a null value indicates the architecture configuration has not been evaluated for likes) [50]. For design decision notes, a corpus and dictionary were designed with a simple grammar originally based on the Lorem Ipsum Python library[5]. A function was created to parse the attribute values to generate notes. If the values were high, the *Agent's* function extracts a "positive" word to describe the attribute. Conversely, if the value is low, the *Agent's* function extracts a "negative" word to describe the attribute. Each attribute-value pair is added as a design decision, effectively creating three decisions per agent per architecture configuration. The scores, like-statues, and design decision notes are then added to a "local *User*" file that is merged with the baseline architecture file (as described in Sec. 6.3.5).

6.4.4 Neo4J Graph Database Plugin

Utilizing Neo4J for UC1 requires several steps. Consider that the initial transforming, scoring, liking, documenting design decisions, and merging user metadata are complete for a single architecture. Using this merged model (in JSON), a query language used by Neo4J, called Cypher, is used to create the GDB. A Cypher generation module parses the merged JSON architecture model for the nodes and relationships (e.g., a User node, an "OWNS" relationship). Nodes have relationships that connect them to other nodes, based on the metamodel (Fig. C.11). The Cypher generation module uses Cypher specifications, like CREATE, to cre-

[5]Lorem Ipsum is a random word/sentence generator library; https://pypi.org/project/lorem/

ate a node or relationship. Parameter values, notes, and metadata are added in Key-Value pairs that match the layer/level in the JSON architecture model.

The module's output is saved as a .cypher file, which can be utilized in any Neo4J application (e.g., desktop, Sandbox, or Oasis). We used the Python API connection drives with the neo4j and py2neo Python libraries and the Bolt communication protocol. The Bolt protocol supports three additional data types: Node, Relationship, and Path (along with core data types like Boolean, integer, or string). Next, a Cypher sender module calls a Neo4J API connection driver function and sends the Cypher query language to a back-end Neo4J process. This process reads the CREATE queries containing the node and relationship data types to build the GDB. Once the GDB is built, queries can return visualizations of the database, its contents, and what metadata is stored at what level.

Using graph data science, the impact of changing a single FireSat subsystem component can be evaluated. For example, a lower performance solar panel can be changed for a higher performance battery which influences a designer's scoring for qualitative attributes, including quality or maturity. The component's physical (architectural) interfaces may change, along with dependencies to specific types of batteries that have varying current/voltage/power requirements. With enough data and traceability to design decisions, GDBs can paint an image of *why* specific attribute scores were given, influencing the like status, ranking, and ultimately the choice to select an architecture configuration or not. Finally, the GDB and metadata can be plotted to visualize the contents of the GDB (Fig. 6.11). For explainability, a class diagram demonstrating the full utilization of the Neo4J Graph Database Plugin system is shown in Fig. 6.12.

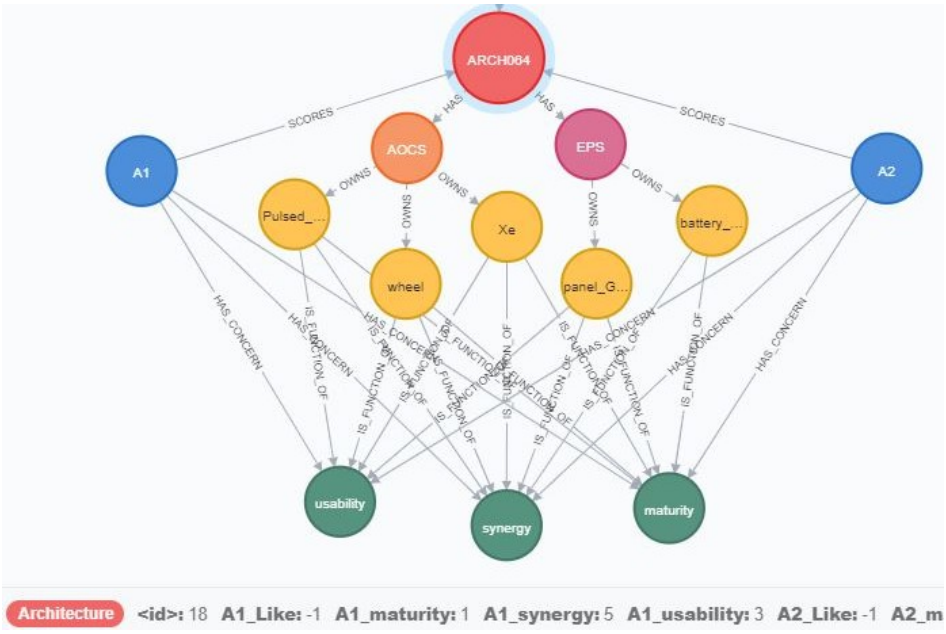

Figure 6.11: FireSat Neo4J Graph Database. This figure shows an example of an architecture (red) with AOCS and EPS subsystems (orange and pink). Subsystem components (yellow) are a function of qualitative/architectural attributes (green). Users (blue) score the architecture with concerns regarding the ascribed attributes.

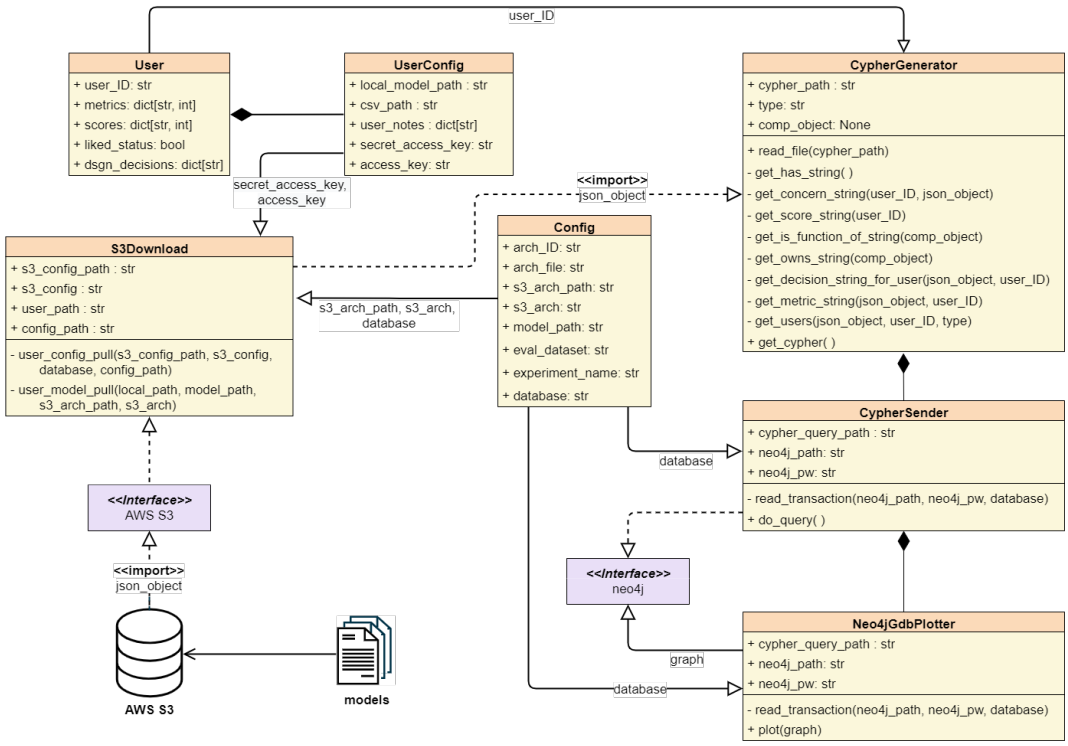

Figure 6.12: Neo4J Graph Database Plugin Class Diagram.

6.4.5 Weights & Biases Experiment Tracking Plugin

For UC2, designing the Weights & Biases (wandb) plugins is similar to the Neo4J plugin construction. The primary difference is that wandb requires data to be sent to its API using either parameter vector form or via a custom dictionary containing all representable parameters and attributes. This dictionary holds attributes and values in a key-value pair. Attributes can be tagged with metadata and notes in a non-numerical (str) format but must be differentiated as a categorical value and not a numerical value. Additionally, wandb uses the HTTPS protocol for communicating through the connect drivers and API. For FireSat, wandb acts as a metadata and model version control and storage for attributes and (hyper)parameters. These model configurations are stored in a .yaml file and can be rebuilt or reevaluated when desired.

Experiment runs (e.g., our architecture configuration models) can be staged either individually or in batches for wandb. This means that the *Orchestrator* can either post experimental data to the wandb storage location as it comes or wait until a number of runs (sprints) have been completed. Additionally, each run will generate a new "version" of the metadata model, even if the model contains the same architecture configuration and parameters. For the FireSat case study, wandb is used in two ways. The first example is comparing multiple runs of the same architecture configuration except that designers have changed their values for qualitative/architecture attributes (Fig. C.12). The second example is when designers want to compare multiple (discrete) architecture configurations and their parameter values

(Fig. C.13). Fig. 6.13 demonstrates a "sweep" comparing five different architecture configurations and their attribute parameter values for quantitative scores (evaluation dataset) and qualitative/architectural attributes (scored by *Agents*).

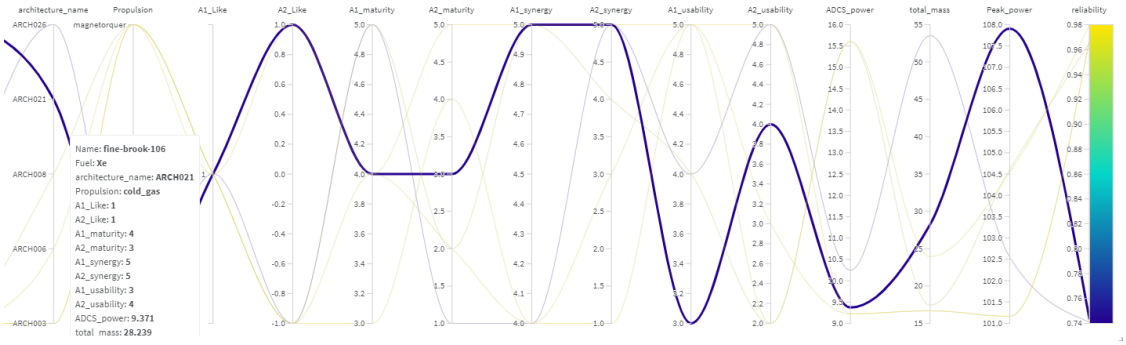

Figure 6.13: Wandb. Five FireSat architecture configurations are compared against qualitative, quantitative, and architecture attributes for two Agents. The box on the left shows metadata associated with the selected model.

6.4.6 Analysis in Discover

The last end-to-end pipeline task is to visualize all architecture configurations that have been scored and liked. Combining these models into a single CSV file is the (currently) required data format for visualizations in Discover, SCS's front-end analysis platform. First, all (merged) architecture configuration models are pulled from AWS into a merged models folder on the *Orchestrator*'s workstation. Next, a "fully-combined architectures" module loops through all architectures stored in the folder and creates a single CSV file. Next, the CSV file is loaded into Discover so that users can visualize trade-offs, use filters, like, and rank architectures. This process leads to another iterative run as described throughout this report until a consensus is made on the preferred architecture configuration candidates to prepare for Phase A studies. A full image of the Discover dashboard with all 64 completed architecture configurations is provided in Fig. B.1 (too large to place in-text).

6.4.7 User-Defined Module Plugins

Extending user-defined modules is relatively simple, following the Plugin creation details in Sec. 6.4.2. In this section, two use-defined modules for UC4 are converted to plugins.

Feature Importance via Correlation

The first example is using correlation matrices to identify feature importance and co-factor confounding. First, the fully-combined architecture model (Sec 6.4.6) was sent to AWS Glue

Databrew[6] to perform One-Hot Encoding (OHE) [51] on all subsystem components (e.g., "NiCa Battery"). OHE provides non-numerical category entries (str) with either a 1 having this property or 0 for missing this property. Having fully numerical data allows us to use statistical analysis algorithms for correlation or autoregression during exploratory data analysis.

Next, a function creates a dataframe from the FireSat dataset. Then, a seaborn[7] function computes the correlation matrix. Afterward, matplotlib[8] is used to return a plot to the SCS Notebook. Additionally, the scikit-learn[9] Feature Analysis package can be used to identify latent patterns by decomposing the correlation matrix. Based on our fully-completed FireSat dataset, we can see that there are strong negative correlations between Reliability and using Xenon fuel. Additionally, *Agent 1* tends to like architecture configurations based on the Usability quality attribute and *Agent 2* tends to like architecture configurations based on the Synergy architecture attribute (Fig. C.15 in Appendix 2).

Designer Sentiment Analysis

The second example uses design decision note metadata stored in the GDB to find user sentiment. Sentiment can help identify what designers like about an architecture, identify how optimistic/pessimistic they score architectures, or predict which future architectures will be liked/disliked. One assumption is that designers are specific and expressive when writing design decisions.

Simulating the sentiment process starts with a connection to Neo4J to access a previously stored GDB containing designer notes. TextBlob[10] is used to extract the design decision notes into the Python environment. Next, the design decision notes are passed through NLTK[11] and sentiment is bounded between -1 and 1 for negative sentiment and positive sentiment, respectively. For example, a sentiment score of 0.75 means the designer's language indicates a good design. Conversely, a score of -0.90 indicates that the designer is highly dissatisfied with the design. Additionally, keyword spotting was implemented to return the primary keywords within design decisions that result in a given sentiment score. Fig. 6.14 shows the input design decision notes and Fig. 6.15 shows the resulting sentiment values for Agent 1.

[6] AWS Glue Databrew is a technology for data cleaning and normalization; https://aws.amazon.com/glue/features/databrew/

[7] seaborn is a statistical data visualization framework; https://seaborn.pydata.org/index.html

[8] matplotlib is data visualization plotting tool; https://matplotlib.org/

[9] scikit-learn is a predictive data analysis library; https://scikit-learn.org/stable/index.html

[10] TextBlob is an artificial intelligence library for processing textual data; https://textblob.readthedocs.io/en/dev/

[11] NLTK is the leading Natural Language Toolkit for building Natural Language Processing (NLP) applications; https://www.nltk.org/

```
([{'A1_DD1': 'This system architecture has excellent synergy and I love it.'}],
 [{'A1_DD2': 'This AOCS model has sufficient usability.'}],
 [{'A1_DD3': 'I hate the absymal maturity of this terrible architecture.'}],
 [{'A2_DD1': 'I am moderately satisfied with the nominal synergy.'}],
 [{'A2_DD2': 'I hate the horrible usability of this terrible EPS system.'}],
 [{'A2_DD3': 'I prefer the great maturity of this design.'}])
```

Figure 6.14: FireSat Agent-generated Design Decision Notes. These notes are extracted with TextBlob and used by NLTK for sentiment analysis.

```
-----------------------------------------------------
Experiment, Architecture: ('cogent-firesat', 'ARCH023')

User: A1

Design Decision 1 - Sentiment: 0.75
 Keywords:['system architecture', 'excellent synergy']

Design Decision 2 - Sentiment: 0.0
 Keywords:['aocs', 'sufficient usability']

Design Decision 3 - Sentiment: -0.9
 Keywords:['absymal maturity', 'terrible architecture']
-----------------------------------------------------
```

Figure 6.15: NLTK Output. The previous three FireSat design decisions for Agent 1 are scored with NLTK's sentiment analysis package. The scores of 0.75, 0, and -0.9 translate to "good," "neutral," and "very poor."

6.4.8 Plugins Execution & Interaction

The final architectural task is to describe the sequences in which events occur. Recall that the plugin manager has three defined order triggers. Fig. 6.16 shows four different parallel branches of plugins. For the Neo4J path containing Make Cypher, Make GDB, and Plot GDB, the ordering of plugin modules is essential. Make Cypher must always fire before Make GDB, or the pipeline can fail. In this case, we force the order using the ordering triggers described in Sec. 6.2.1. Plugins not assigned a timing module will trigger with a "Last In First Out" scheduling protocol. Strict plugin behavior can be assigned in the plugin manager, but it is not desired to remove the abstraction level of the plugin manager and cause potential tangling of resources. Changing the plugin manager's core program should occur as little as possible.

Now, the system architecture and design have been covered. For a final FireSat functionality description diagram containing all relevant technologies and plugins, please see Fig. B.2. The following section will cover the verification and validation aspects more specifically.

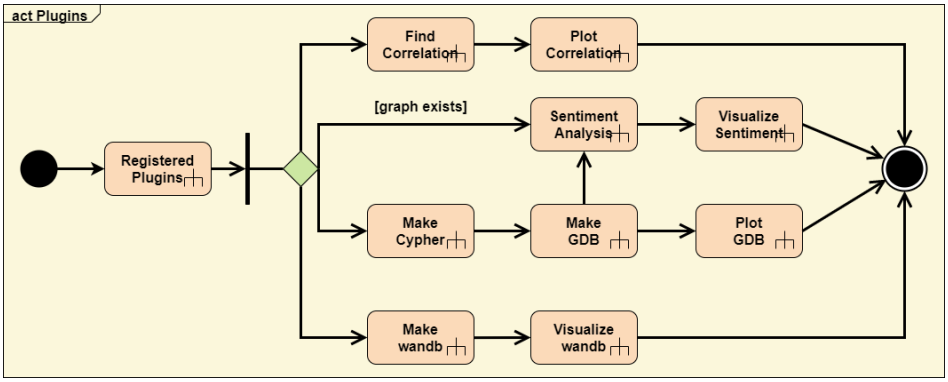

Figure 6.16: Plugin Activity Diagram. Four different plugin paths occur in parallel, but with forced timings based on plugin ordering specifications.

7 Verification & Validation

This chapter details the verification and validation aspects for COGENT. Additionally, this chapter contains the testing phases and *Agent* ranking module. For this report, the IEEE-STD-610 verification and validation definitions are used.

7.1 Verification

Verification confirms if the requirements/specifications are correctly implemented. For this report, verification is split into the functional requirement verification and system/software testing phases.

7.1.1 Functional Requirement Verification

There were 57 formal functional requirements derived for the general software system and COGENT. A complete list of functional requirements and their results are included in Fig. B.2. Sec. 5.2 lists the 16 primary functional requirements, with 14 requirements being achieved and 2 requirements being partially achieved. The two partially achieved requirements, *R1.2 - the integration of ranking results from multiple users*, was transferred back to SISW as the Ranking functionality was not implemented in Discover's front-end at the time of the COGENT project. Similarly for *R7.2 - The system shall automatically rank configurations based on quantitative/qualitative/architectural scores*, a weighted ranking ([52], [53]) module was created but will not match the actual direction SCS/Discover will use for ranking architectures (ranking is purely analytical in Discover).

Ranking Module

Effectively, the partial realization of the Ranking activity (R1.2, R7.2) is a plugin for the autonomous *Agents*. A weighted ranking function first checks to see if all *Agent's* have "liked" as a parameter value in the liked status, e.g., "A1_like = 1" means *Agent 1* likes the architecture based on the process in Sec.6.4.3. Next, dictionaries are constructed with weights for the quantitative attributes and qualitative/architectural attributes in a key-value format. For Fig. 7.1, the quantitative dictionary is "'Reliability': 0.4, 'Total_mass': 0.1, 'ADCS_power': 0.2, 'Peak_power': 0.3", such that weights are normalized to sum to 1 and the weights of the quantitative attributes indicate the *User's* preferences.

architecture_name	Total_mass	ADCS_power	Peak_power	Reliability	A1_Like	A2_Like	A1_synergy	A1_usability	A1_maturity
ARCH021	28.239286	9.370770	107.908293	0.743274	1	1	5	3	4
ARCH055	26.519400	10.394620	165.479978	0.648108	-1	1	4	2	5
ARCH049	31.681120	9.921132	156.038238	0.657641	1	-1	4	5	3
ARCH062	24.571548	10.020085	133.457088	0.655641	-1	1	2	3	1
ARCH061	29.006803	10.023827	133.458916	0.657641	-1	1	4	1	3
...	...	...	...	...	...	...	...	...	...
ARCH027	51.812343	10.289428	102.587979	0.732500	-1	-1	4	1	3
ARCH041	52.882816	10.813259	102.199052	0.657641	-1	-1	2	1	5
ARCH025	57.957365	10.251815	102.610578	0.743274	-1	-1	4	3	1
ARCH020	99.774893	10.464980	102.840639	0.730270	-1	-1	2	2	3
ARCH017	110.401554	10.507740	102.926441	0.743274	-1	-1	2	4	3

Figure 7.1: Simulated Ranking Results. ARCH021 provides the best trade-offs with both *Agent's* having "liked" the architecture configuration.

With this setup, "ARCH021" has the highest rank with two *Agent* likes, low total mass, a good trade-off between AOCS power and peak (system) power. The architecture attribute value for Synergy is good with overall high qualitative attribute scores and a decent reliability score. This approach was inspired by design-task-oriented value assignment for MBSE [54] and multi-attribute trade-space exploration [55].

7.1.2 System/Software Testing

The system was testing iteratively in three phases, including Unit, Integration, and Functional. A Test-Driven Development (TDD) approach was taken [56]. Additionally, common "best practices" and common Python-specific anti-patterns were followed [57]. TDD describes developing code around testability and introduces a Model-Conductor-Hardware design pattern, in which system development is on "mock" hardware (simulated/emulated). This concept will become more important in a few paragraphs regarding the user simulation agents.

Unit Testing

First, each software module had unit tests constructed in PyTest[1]. Unit tests focused on invalid data types and asserting null or out-of-range values in function arguments. Testing for null values was critical since the design process allows users to add data models in cycles and out of order. The methods/functions compiling query languages, dictionaries, or merging files were extended with Python's standard exception handling to detect missing/null values and tell the program to continue building the model with the proper format ("try, except"). Furthermore, the plugin setup files contain an option to flag unit tests for continuous/automatic testing with PyTest. In parallel, we used PyCharm's[2] built-in code inspections for static analysis on the completed module. A JSON validation package was used to validate architecture configuration models against required metamodel structure specifications [58].

[1] PyTest is the core unit testing framework for Python; https://www.pytest.org

[2] PyCharm is a popular Integrated Development Environment for Python; https://www.jetbrains.com/pycharm/

Integration Testing

Unit-tested software modules with shared interfaces require integration testing. Data/content is input into the first module and checked as output for the second module. Using PyCharm to "step-through" the code, variable values and data types can be viewed after each process, transformation, or algorithm. Integration testing was performed in "chains" by adding the subsequent (unit-tested) software testing pipeline. End-to-end integration thoroughly applied to 64 system architecture configurations with a minimum of five times each. End-to-end integration starts from the initial JSON model generation to the final third-party visualization tooling manual inspection, with AWS communications in between.

Functional Testing

For this project, functional testing acts from a "super-integration" perspective. After the end-to-end integration was error-free, we integrated and tested each module into the plugin manager (individually). One challenge with plugins is that they may have asynchronous firing, which breaks the system pipeline. Debugging is more challenging using a plugin architecture, as a plugin might be invoked with a deadlock whenever the plugin manager communicates with programs outside of the closed-loop environment. As such, functional testing required substantially more time than unit or integration testing. From within the COGENT system, objects stored over APIs were queried to ensure objects had correctness and were not empty (e.g., SQL for AWS S3).

Multi-Environment Testing

To simulate the collaborative and iterative design process, an *Orchestrator* environment ran in parallel with *User* mock environments. SCS is built based on Jupyter Lab[3], and other types of Notebooks, including Google Colab[4] are interoperable with SCS. COGENT and all plugins work in SCS, Jupyter Lab, and Colab (when using a Python 3 kernel). A majority of the multi-environment COGENT testing had the *Orchestrator's* environment in SCS on one machine and the *Users'* environments in Colab on another machine. This setup ensured that cross-machine communication and plugin behavior was correct, as Notebooks store global data and module output data in a local cache. The entire multi-environment pipeline was tested for all 64 architecture configurations simultaneously with one *Orchestrator*, three *User's*, and 19 concurrent plugins (no data collisions).

7.2 Validation

Validation confirms if the product fulfills the intended use and goals of the stakeholders. Validation was approached with demonstrations, regular feedback, cross-domain analogies, technological realizations, and non-functional requirements.

[3]Jupyter Lab is a browser version of the most popular Computation Notebook; https://jupyter.org/

[4]Google Colab is a Google-based Computation Notebook environment for sharing and co-developing Notebooks; https://colab.research.google.com/

7.2.1 Demonstrations

Early in the project, SISW confirmed that *tangible results and examples* were the most critical aspect of the COGENT project. Feature and technology demonstrations were given every two weeks to guide the validation process and ensure the right product was developed. The end-user was the priority throughout the product, and we tried to identify how to provide them with the best solutions and most straightforward implementation. Weekly meetings discussed feature and architectural possibilities.

7.2.2 Cross-Domain Analogies

Siemens SISW already has experience in applying generative engineering in the automotive domain. One critical step was applying the COGENT framework to previous work. For example, we created a GDB in Neo4J based on an architecture configuration previously created using SCS/Amesim (Fig. C.14). This step was taken to ensure models with proper relationships could be created for GDBs. At a high level, many CPSs are a SoI with subsystems, components, parameters, and relationship classes. With a high-level domain metamodel, these specific classes swap identifies and discrete values (e.g., "a vehicle HAS a power subsystem" is similar to "a satellite HAS an EPS subsystem").

7.2.3 COGENT Technology Realizations

As described with CAFCR's Realization Viewpoint (Sec. 6.4), specific technology solutions we developed for the FireSat case study. Each of these demonstrations was built using the COGENT plugin system. *Users* were simulated as *Agents.* Naturally, simulations do not provide a full perspective into how an end-user will really interact with the system. We have shown that end-users can use their preferred technology or import the current (Python) modules with ease. Plugins can be developed for other programming languages and integrated with Python wrappers, but this was outside the project's scope. Each UC and plugin example was created with CDF end-users in mind, trying to understand how and why COGENT could improve the concurrent engineering design process and communication for cross-functional teams.

7.2.4 Non-Functional Requirements

Throughout the project, several quality attributes (non-functional requirements) were pursued to provide a complete and robust solution. Of 13 quality attributes, 9 were identified as important for either end-users or Siemens SISW (medium and high priority). A complete list of non-functional requirements is contained in Sec. B.1. Four categories based on ISO/IEC 25010:2011 are provided below: Usability, Security, Modularity, Explainability, Functional Suitability, and Portability. Table 7.1 shows nine mid/high priority non-functional requirements and their achieved statuses.

Table 7.1: COGENT Non-Functional Requirements. The Priority (P) is shown as High (H) or Medium (M). The color encoding shows achieved (green) or partially achieved (yellow).

ID	Concern	P	Description
NFR1	Usability	M	The system shall support conflict resolution in decision-making sessions (align discussion).
NFR2	Usability	H	The system shall support the balancing of multi-user expectations/needs.
NFR3	Usability	H	The system shall be as autonomous as possible.
NFR4	Modularity	H	The system shall be developed in an independent manner, not imposing dependencies on SCS.
NFR5	Usability	H	The system shall be developed based on approved licenses (e.g., open source)
NFR6	Explainability	H	The system shall be provided with a thorough ConOps and complete documentation.
NFR7	Usability	M	The system shall address the major individual "pain" for subsystem teams.
NFR9	Portability	H	The system shall be easy to install
NFR10	Suitability	M	The system shall differentiate between quantitative, qualitative, and architectural FoMs.

Usability

Of the nine mid/high priority NFRs, five of these requirements fall under Usability. NFR1 and NFR2 were only partially achieved as while the COGENT system was designed to support conflict resolution and balance multi-user expectations, we do not have any quantitative or qualitative results to verify this. Similarly, NFR7 is responsible for addressing the major "pain" of a subsystem team, yet this was achieved by having specific quantitative, qualitative, and architectural attributes addressed during every process of the pipeline. NFR5 requires all code and software to be built in a way that does not reduce the accessibility of end-users or SISW since licensing can become a significant financial or legal dilemma. NFR3 is related to automatability and will be discussed more next.

Automatability

The architecture and data models were developed with a high level of abstraction to enable system components to be automatable. Some fixed relationships exist between the general system architecture, the user, the subsystems, and the components. For example, a user always scores an architecture, and an architecture configuration always contains subsystems and components. Using this knowledge to develop schema, the created data model files and interfaces do not need to be changed to connect with different databases of a similar type (i.e., creating Cypher and AQL queries in Neo4J and ArangoDB uses the selfsame base model file) [8]. The same is true for the automatic sharing of configuration files such that the system automatically assigns the *User* with the architecture configuration ID selected by the

Orchestrator. This process ensures that designers will never score an architecture that does not exist and can only score attributes/FoM that exist (Keys; compliance to SEIM).

Furthermore, each plugin will automatically run whenever it has been registered with the plugin manager. During testing, a set of *Agent's* could complete an entire "design sprint" in less than 10 minutes, from the initial model transformation to storing models in a GDB and visualizing the result. While *Users* will undoubtedly require more time, a majority of "thought-less," mundane tasks in the design process have been bypassed. Furthermore, this will result in fewer errors overall and allow *Users* to focus on the essential tasks without system-based interruptions.

Modularity

The description of NFR4 describes how COGENT should be independent and not impose dependencies on SCS. Independence was a major motivation for using a plugin architecture pattern. *Users* install COGENT directly into their SCS environment without COGENT affecting any other installed package or service. The COGENT plugin manager is executable from Notebook cells or the command line. Plugins can be deregistered simply by removing the entry point. In general, the example plugins were designed to be independent and even agnostic of each other. However, co-interactions are possible if the *User* desires as described in Sec. 6.4.7.

Portability

Installing the COGENT system was designed to be easy (NFR9). The *User* simply needs to add the COGENT plugin manager and any pre-defined plugins to their SCS environment by either loading a folder or cloning a Git repository. The plugin setup files were designed to install any Python libraries/dependencies automatically and automatically use PyTest for unit tests (given that test cases are provided). Additionally, specific versions of software packages/libraries are declared in the plugin setup file, ensuring the modules work as intended.

Explainability & Functional Suitability

Developing documentation is an important aspect of product hand-over and usage (NFR6). We provided an installation guide, a user guide, and code comments following the PEP-8 Style Guide for this project. Furthermore, video guides demonstrate tasks such as installing COGENT, installing plugins, and creating plugins. Furthermore, a Concept of Operations (ConOps) was a required deliverable for SISW. The ConOps specifies the system's environment, usage examples, and stakeholder needs on a high-level [14]. Please see Sec. C.7.1 for more details on the ConOps. Finally, a functional suitability requirement (NFR10) was formally defined to ensure that the system differentiates between qualitative, quantitative, and architectural attributes.

The next chapter provides the conclusion and recommendations.

8 Conclusion & Recommendations

This chapter briefly summarizes the results of the project and report with recommendations for future work.

8.1 Conclusion

This report details the path to develop a solution architecture and proof-of-concept implementation applying generative engineering in the concurrent engineering domain. System architecture selection is a challenging and wickedly complex task. Subsystem designers have conflicting goals and requirements that impact technological capabilities and system performance. Having the most powerful subsystem/component is going to have immediate trade-offs with costs, mass, life expectancy, and interfacing with other components (interoperability). Furthermore, selecting an inferior architecture configuration is detrimental to system performance, costs, and life-cycle, with the additional risk of mission/experiment failure. Both concurrent and generative engineering methodologies significantly improve system architecture design during the exploratory and concept design phases. However, a union of the two methodologies had yet to be realized with demonstrable and tangible technologies.

The decision was to create a plugin architecture capable of transforming data/models, storing models, assets, and metadata in desired storage locations, while also developing plugins to demonstrate technology capabilities. A conceptual FireSat case study was followed where cross-functional teams in a CDF attempt to select an optimal system architecture configuration. It was shown that metadata is a critical component in the design decision process, and how/where the metadata is stored is important. Additionally, end-users now have access to third-party technologies and their own user-defined modules within Siemens SCS. We think this will encourage users to utilize SCS for concurrent generative engineer and design in the aerospace, automotive, and robotics domains. The project and report focus on providing understanding end-users needs and demonstrating useful technologies as a solution.

Furthermore, the importance of differentiating quantitative, qualitative, and architectural attributes emerged. Each attribute type has varying parameters and implications for the design process. Quantitative attributes are easier to handle since parameters are objective (based solely on numerical criteria). For qualitative and architectural attributes, any two designers may have drastically different perspectives based on their primary concerns, experience, and goals. Users can definitively represent such concerns in our metamodel structure and technology demonstrations. We hope COGENT will improve cross-functional team communication, explainability and traceability of design decisions, and a more rapid system architecture configuration selection with higher performance indicator results.

8.2 Project Results

In the introduction, we described six high-level requirements. All six high-level requirements were achieved with the following details:

HLR1: SCS was extended from a single-user user application to a multi-user application with simultaneous tool access

HLR2: An end-to-end plugin architecture was demonstrated combining concurrent and generative engineering domains

HLR3: The user scoring and ranking activities were thoroughly simulated and had consistent, synchronized results

HLR4: Four use cases utilizing external or custom software were demonstrated for the FireSat case study, including graph databases, experiment tracking, centralized cloud storage, and user-defined modules

HLR5: Metadata was used for tracking, reporting, analysis, and discussing design decisions at the system, subsystem, and component-level with both internal and external software demonstrating the significance of metadata

HLR6: A centralized AWS Cloud solution was used to store objects, assets, metadata, and models; with the possibility to easily switch the storage solution by changing the API

8.3 Recommendations

The most significant limitation of this study and project is the lack of evaluation by human designers in a real case study. We recommend that COGENT is demonstrated and implemented in an actual concurrent engineering case study in a concurrent engineering environment. Intuitively, an aerospace/astronautic example would be the easiest to evaluate since COGENT was designed for an analogous FireSat case study. However, any SoS CPS domain with subsystems, components, attributes/parameters, and users can be used.

Additional functionalities and plugins could be developed as part of a "core COGENT" distribution. Siemens already creates tools for finite element analysis, computational fluid dynamics, life-cycle management, and computer-aided engineering. Secondary technologies might include the generation of domain specific languages, automated machine learning (AutoML), synthetic data generation, or digital twin environments. Each of these topics are up-and-coming in the space domain, as well as other Industry 4.0 domains. Ideally, running a plugin that communicates to these technologies could return simulation results. Future COGENT developments may consider interfaces and wrappers to communicate with such technologies.

Bibliography

[1] Daniel Hastings and H. Mcmanus. Space system architecture: Final report of ssparc: the space systems, policy, and architecture research consortium (thrust i and ii). 01 2005.

[2] Warschat, J. and Bullinger, H.J. *Forschungs- und Entwicklungsmanagement, Simultaneous Engineering, Projektmanagement, Produktplanung, Rapid Product Development*. 1997. doi:10.1007/978-3-663-05946-2.

[3] G. Muller. CAFCR: A Multi-view Method for Embedded Systems Architecting. Balancing Genericity and Specificity. 06 2004.

[4] Daarius user manual. Technical report, TNO-ESI, 2021.

[5] M. Bandecchi, B. Melton, and F. Ongaro. Concurrent Engineering Applied to Space Mission Assessment and Design . Technical report, European Space Agency, 1999.

[6] European Space Agency. What is the CDF?, Accessed: April, 2020. http://www.esa.int/Space_Engineering_Technology/CDF/What_is_the_CDF.

[7] O. Alotaibi and E Pardede. *Transformation of Schema from Relational Database (RDB) to NoSQL Databases*, volume 4. 11 2019. 10.3390/data4040148.

[8] Hunger, M. and Boyd, R. and Lyon, W. *The Definitive Guide to Graph Databases for the RDBMS Developer*. 2021.

[9] M.W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998.

[10] M. Bandecchi, B. Melton, and B. Gardini. The esa/estec concurrent design facility. 01 2000.

[11] Churchman, C.W. *"Wicked Problems". Management Science. 14 (4): B-141–B-146.* December 1967. doi:10.1287/mnsc.14.4.B141.

[12] J. Fitzgerald, P. Larsen, and M. Verhoef. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. 05 2014.

[13] A. Kusiak. *Concurrent Engineering: Automation, Tools, and Techniques*. A Wiley-Interscience Publication. Wiley, 1992.

[14] *NASA Systems Engineering Handbook, NASA SP-2016-6105 Rev2*. October 2017. https://www.nasa.gov/connect/ebooks/nasa-systems-engineering-handbook/.

[15] J. Hall. Columbia and challenger: Organizational failure at nasa. *Space Policy*, 37:127–133, 08 2016.

[16] W. Robison, R. Boisjoly, D. Hoeker, and S. Young. Representation and misrepresentation: Tufte and the morton thiokol engineers on the challenger. *Science and engineering ethics*, 8:59–81, 02 2002.

[17] W.J. Larson, J.R Wertz, and B. D'Souza. *Space Mission Analysis and Design*. 09 1999.

[18] A. Mincolla, S. Gerené, A. Vorobiev, K. Wojnowski, N. Smiechowski, M. Nicolai, J. Menu, S. Jahnke, R. Benvenuto, and G. Tibert. Space systems of systems generative design using concurrent mbse: An application of ecss-e-tm-10-25 and the gcd tool to copernicus next generation. pages 1–6, 2020.

[19] A. Silva. Model-driven engineering: A survey supported by a unified conceptual model. *Computer Languages, Systems & Structures*, 20, 06 2015.

[20] J. Menu, M. Nicolai, and M. Zeller. Designing fail-safe architectures for aircraft electrical power systems. In *2018 AIAA/IEEE Electric Aircraft Technologies Symposium (EATS)*, pages 1–14, 2018.

[21] W. Quan and H. Jianmin. A study on collaborative mechanism for product design in distributed concurrent engineering. In *2006 7th International Conference on Computer-Aided Industrial Design and Conceptual Design*, pages 1–5, 2006.

[22] European Space Agency. ESA Concurrent Design Facility CDF Infopak, Accessed: August, 2020. https://esamultimedia.esa.int/docs/cdf/ESA_CDF_Infopack_2019_16x9_rev1.pdf.

[23] ECSS-E-TM-10-25 - System Engineering - Engineering Design Model Data Exchange (CDF). Technical report, European Cooperation for Space Standardization, October 2010. https://ecss.nl/hbstms/ecss-e-tm-10-25a-engineering-design-model-data-exchange-cdf-20-october-2010/.

[24] DoDAF V2.02, Volume II: Architectural Data and Models, Architect's Guide . Technical report, Department of Defense, January 2015.

[25] P. Kruchten. Architectural blueprints—the "4+1" view model of software architecture. *IEEE Software 12*, 6:42–50, 11 1995.

[26] G. Muller. *Systems Architecting: A Business Perspective*, volume 21. 06 2011.

[27] DAARIUS for Transparent Team-based System Design. 11 2019.

[28] T. Bijlsma, W.T. Suermondt, and R. Doornbos. *A Knowledge Domain Structure to Enable System Wide Reasoning and Decision Making*. apr 2019. 10.1016/j.procs.2019.05.081.

[29] T. Bijlsma, B.v.d. Sanden, Y. Li, R. Janssen, and R. Tinsel. Decision support methodology for evolutionary embedded system design. In *2019 International Symposium on Systems Engineering (ISSE)*, pages 1–8, 2019.

[30] *ESA PSS-05-02 Guide to the user requirements definition phase*. March 1995. http://microelectronics.esa.int/vhdl/pss/PSS-05-02.pdf.

[31] Robinson, I. and Webber, J. and Eifrem, E. *"Graph Databases". 2nd Edition.* June 2015. ISBN: 9781491930892.

[32] L. Lazarevic. Identifying graph shaped problems, 2020. Access Online: https://go.neo4j.com/identifying-graph-shaped-problems.

[33] Experiment tracking, Accessed: 20.10.2021. https://neptune.ai/experiment-tracking.

[34] L. Biewald. Experiment tracking with weights and biases, 2020. Software available from: www.wandb.com.

[35] N. Franke, P. Keinz, and C. Steger. Testing the value of customization: When do customers really prefer products tailored to their preferences? *Journal of Marketing American Marketing Association ISSN*, 73:103–121, 10 2009.

[36] A. Achilleos, N. Georgalas, and K. Yang. An open source domain-specific tools framework to support model driven development of oss. volume 4530, pages 1–16, 06 2007.

[37] *ESA PSS-05-03 Guide to the software requirements definition phase*. March 1995. http://microelectronics.esa.int/vhdl/pss/PSS-05-03.pdf.

[38] Nadareishvili, I. and Mitra, R. and McLarty, M. and Amundsen, M. *Microservice Architecture*. August 2016. ISBN: 9781491956250.

[39] M. Richards. *Software Architecture Patterns*. February 2015. ISBN: 9781491924242.

[40] D. Namiot and M. sneps sneppe. On micro-services architecture. *Interenational Journal of Open Information Technologies*, 2:24–27, 09 2014.

[41] J. Mayer, I. Melzer, and F. Schweiggert. Lightweight plug-in-based application development. 09 2002.

[42] Richards, M. and Ford, N. *Fundamentals of Software Architecture*. January 2020. ISBN: 9781492043454.

[43] Pluggy: The PyTest Plugin System, 2020. Software available from: https://pypi.org/project/pluggy/.

[44] Gamma, E. and Helm, R, and Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. October 1994. ISBN: 0201633612.

[45] Raj, P. and Raman, A. and Subramanian, H. *Architectural Patterns*. December 2019. ISBN: 9781787287495.

[46] J. Bézivin. Model driven engineering: An emerging technical space. volume 4143, pages 36–64, 01 2005.

[47] D. Kaul, A. Gokhale, L. Dawson, A. Tackett, and K. Mccauley. Applying aspect oriented programming to distributed storage metadata management. 211, 03 2007.

[48] P. Tarr, H. Osccher, W. Harrison, and S. M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. *ICSE '99: Proceedings of the 21st international conference on Software engineering*, page 107–119, 05 1999.

[49] M. Héder. From nasa to eu: the evolution of the trl scale in public sector innovation. *Innovation Journal*, 22:1, 08 2017.

[50] S. Damer and M. Gini. Cooperation without exploitation between self-interested agents. *Advances in Intelligent Systems and Computing*, 194:553–562, 01 2013.

[51] Harris, D. and Harris, S. *Digital Design and Computer Architecture, 2nd Edition*. August 2012. ISBN: 9780123978165.

[52] W.Y. Chiu, S.H. Manoharan, and T.Y. Huang. Weight induced norm approach to group decision making for multiobjective optimization problems in systems engineering. *IEEE Systems Journal*, 14(2):1580–1591, 2020.

[53] D. Urbig. Weight-based negotiation mechanisms: Balancing personal utilities. *Fundam. Inform.*, 67:271–285, 01 2005.

[54] W. Xiaofei, W. Liao, Y. Guo, D. Liu, and W. Qian. A design-task-oriented model assignment method in model-based system engineering. *Mathematical Problems in Engineering*, 2020:1–15, 08 2020.

[55] Jian Guo. Incorporating multidisciplinary design optimization into spacecraft systems engineering. 01 2010.

[56] M. Karlesky, W. Bereza, and C.B. Erickson. Effective test driven development for embedded software. pages 382–387, 05 2006.

[57] A. Dewes and C. Neumann. The little book of python anti-patterns, Accessed: 17.09.2021. Available from http://docs.quantifiedcode.com/python-anti-patterns/.

[58] JSON Format Validation, Accessed: 19.09.2021. Available from https://jschon.readthedocs.io/en/latest/examples/format_validation.html.

[59] J. Raiturkar. *Hands-On Software Architecture with Golang*. December 2018. ISBN: 9781788622592.

A Appendix I - Project Management

This section briefly details the project management aspects of the project. This includes the project stakeholders, project management approach, project timeline, risk management, and a brief retrospective.

A.1 Project Stakeholders

This section describes the stakeholders relevant to the Project Viewpoint. For Product Viewpoint stakeholders, please refer to Sec. 4.

This project was an industry collaboration between the Eindhoven University of Technology (TU/e) and Siemens Digital Industries Software (SISW). Each entity has different concerns based on its primary position and goals. Table A.1 lists the persons, positions, and concerns based on the primary locale. The trainee is at the intersection of both the TU/e and SISW but is primarily a member of the TU/e.

Table A.1: Primary Project Stakeholders. The concerns are from the perspective of the Trainee and may inaccurately represent the stakeholder internal concerns.

Locale	Name	Position	Concerns
TU/e	**C. O'Hara**	**ST Design Trainee**	**Knowledge, skill acquisition, MBSE/MBSA experience**
	M.v.d. Brand	**ST Scientific Director**	**Scientific value, industry valorization, program quality**
	Y. Dajsuren	**ST Program Director**	**Program/project quality, industry valorization, trainee growth**
SISW	**J. Menu**	**Research Engr. Manager**	**Project quality, product utility, stakeholder valorization**
	M. Nikolai	**Sr. Product Line Manager**	**Product usefulness, product relevance, concept clarity**
	J. Vanhuyse	**Product Manager**	**Product development, product usefulness, product testing**

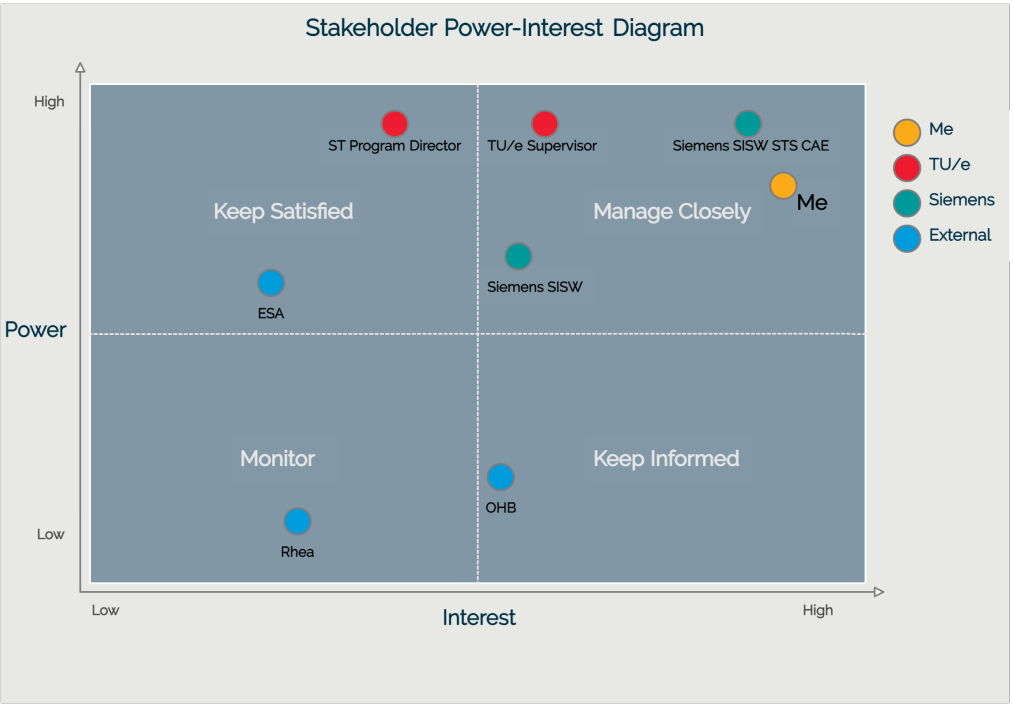

Figure A.1: Stakeholder Power-Interest Matrix Diagram.

A.2 Project Management Approach

Project phases, risk management, phases, and work breakdown structures were derived from ECSS-M-ST-10C, Space Project Management, Rev.1:2009. This project management standard follows the V-Model for the system development lifecycle. For software development, the Agile methodology was followed with software features developed and accessed in weekly sprints. Testing, verification, and validation occurred throughout the project and not in a single phase. A work breakdown structure organized work packages based on the phase and type.

A.3 Project Timeline

The project includes three milestones: the SW Design Freeze, the SYS Design Freeze, and the Project Defense (which assumed the project was complete, tested, verified, and validated). In reality, the dates provided for the Gantt chart in Fig. A.3 were mere approximations as the conceptual and innovative origin of the project required more flexibility in deadlines.

A.4 Risk Management

A risk matrix was derived based on ECSS-M-ST-10C which plots likelihood and severity combinations with color encoding with green, yellow, and red for "good," "caution," and "warning," respectively. Additionally, the gradient of red indicates how potentially disruptive the risk will be on the success of the project. Furthermore, the evolution of risks were tracked throughout the project. Risk management is not something that should simply be completed at the beginning of a project or emerging/evolving risks may not be identified. Fig. A.2 provides the adapted risk management matrix with an example of risk tracking.

Figure A.2: Tracking Risk Evolution in a Risk Matrix. The dotted star is the original approximation of the risk likelihood-severity combination. The dotted arrow shows the transition of the risk to a solid star where the risk currently is located.

A number of risk management strategies were employed including monitoring, accepting, transferring, avoiding, and mitigating. Please see the risk register in A.5 for more details.

A.5 Project Retrospective

The COGENT project was both challenging and rewarding. I feel like I really developed my model-based system engineering/architecting skills to create a meaningful product. Both of my project supervisors allowed me to be creative and exploratory with the design and features. I have been interested in developing system/software solutions for the space domain since I was a child. Additionally, I have been interested in working at Siemens for more than ten years. I feel incredibly fortunate that I could explore all of my interests in my preferred domain at my preferred company.

That being said, the project was still very challenging. Cutting-edge innovations often have many uncertainties, unclear directions, and rapidly changing and/or conflicting requirements.

At the start of the project, the development platform (SCS) had not been announced or released to the public. Therefore, many things were being updated or added regularly. Additionally, we needed to frequently dive into low-level details and then bounce back up to a high-level overview of how this provides value to the end-user. Zooming in and zooming out was more challenging than I anticipated. Furthermore, there were times when I knew how a feature or design alternative could benefit the end-user, but I had trouble explaining this clearly. I feel that I became a domain expert on combining generative and concurrent engineering for space system architectures. Thank you.

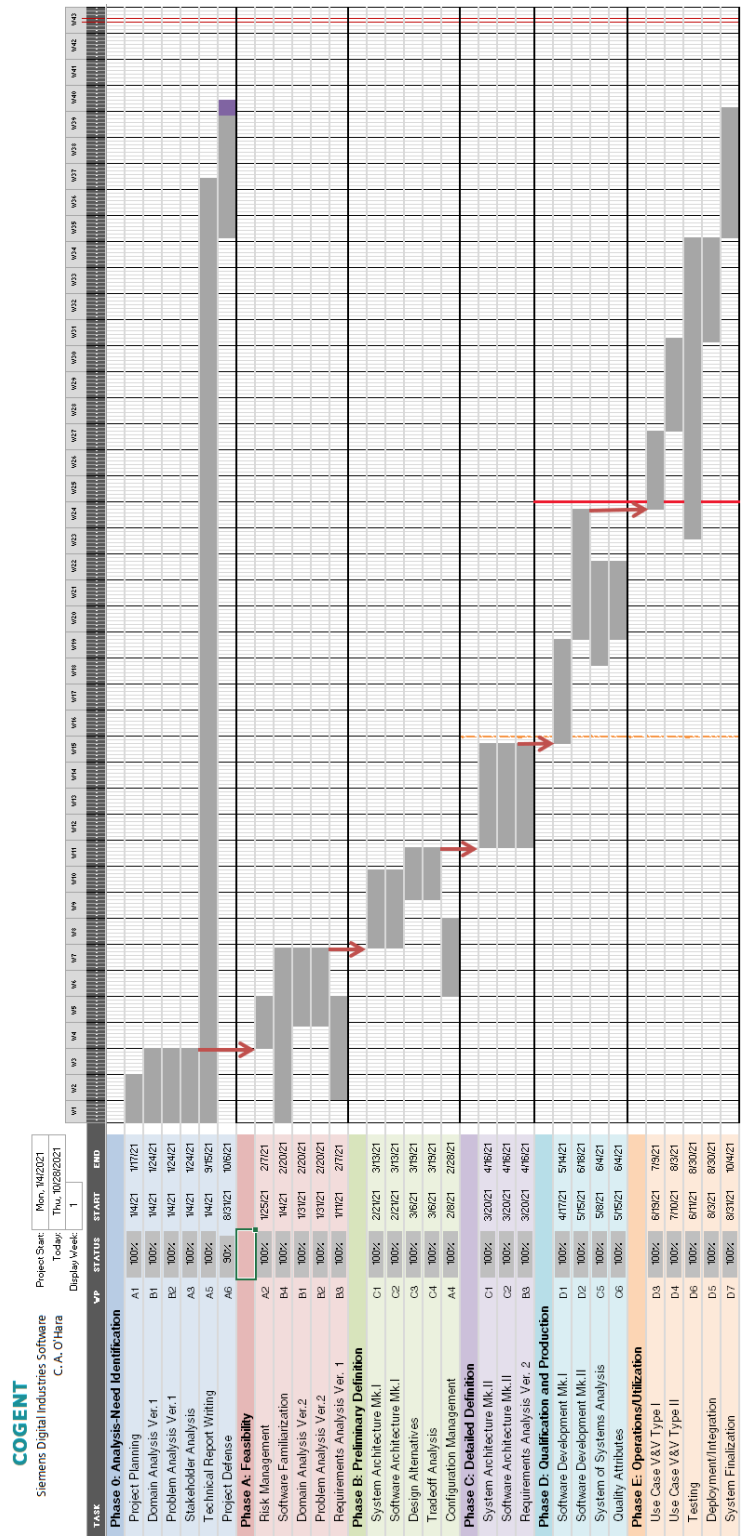

Figure A.3: Project Timeline Gantt Chart.

Risk Register

Project: COGENT		Initial		Risk Approach		Managed	
No.	Risk Scenario	$Severity_1$	$Likelihood_1$	Strategy	Actions	$Severity_2$	$Likelihood_2$
S1	COVID-19 Logistics	3	E	Accept, Monitor	The pandemic impacts the logistics of the project (e.g., company laptop). Currently, AWS instances are being used to work remotely. However, not all software can be ran through proxy/kernals. Relocation will occur if necessary.	1	C
S2	COVID-19 Communications	2	D	Accept, Monitor	Colocation communication, including "coffee break" discussions are lost. Currently, adhoc communication is regularly persued. Supervisors might not be available at a critical moment (design decision/alternative) which can lead to delays in development.	1	C
S3	Administrative Delays	4	D	Transfer, Monitor	Administrative challenges, e.g., contract agreements, lead to delays since company/university IP are limited. The current strategy is to approach the project from a conceptual level and address the high-level and key issues abstractly. Regular communication regarding the contract is maintained.	2	D
S4	Insufficient Domain Knowledge	5	C	Mitigate, Monitor	COGENT emcompasses several complex domains including generative engineering, concurrent engineering, space engineering, system engineering, and software engineering. The project is at the intersection of these domains. Currently, regular communication with company and university supervisors is maintained to ensure proper scoping and domain understanding.	4	A
S5	Insufficient Technical Skills	5	C	Mitigate, Avoid, Monitor	COGENT will potentially be comprised of a complex software implementation as multiple users and stakeholders should be able to interact concurrently. To mitigate insufficient technical skills, prototyping is being completed early (along with training). As a contengency, certain libraries/approaches will be avoided if they introduce unnecessary complexity. Knowledge of DSLs (ACEL), concurrency in Python, data science dashboards, and artificial intelligence is required.	4	A
S6	Security: SW Communication	4	C	Transfer, Monitor	Within the scope of COGENT is to extend a notebook implementation (SCS) to a multi-user setup. However, this will introduce the requirement to communicate over various channels. While SCS requires a key, deployments using Heroku (or equivalent) may allow users to access the notebooks/data without logging into SCS and depending on the deployment environment. The risk here is two part: implementing an insecure environment versus additional resources in developing a secure channel. Currently, Heroku is planned for prototyping and the risk will be transfered later when a production-level implementation is pursued. Since COGENT is a prototype, this risk has a lower severity. The likelihood of an issue is also reduced as security issues will be transfered when COGENT is made at the production level.	2	B
S7	Security: Database	4	C	Transfer, Monitor	Similar to S6, data needs to be securely stored. For prototyping, MongoDB may be used to store data. However, there are limitations to how much data can be stored securely and at no cost. Siemens has an internal database solution though it is not readily accessible. Therefore, this is a risk that will need to be handled with care in the future. The ideal solution to solve S6 and S7 is to use a private server within Siemens that SCS can interface with. As this is not readily available, there is considerable risk that the implementation (prototype) will need to be modified greatly to acheive production-level in the future. Since COGENT is a prototype, this risk has a lower severity. The likelihood of an issue is also reduced as security issues will be transfered when COGENT is made at the production level.	2	B
S8	System Integration	4	C	Mitigate, Monitor	The COGENT project will be developed within (extends) the Siemens Simcenter Studio (SCS) tool. Furthermore, the COGENT application will act as an intermediate step within SCS. As such, the developed features/implementation will have dependencies on SCS and the output will need to conform with SCS. Ideally, the feature additions (e.g., interactive dashboard, scoring/ranking modules) will easily integrate due to the Jupyter Lab aspects of SCS. However, the COGENT implementation may have external calls/functionality that cannot trivially/initially be accessible within SCS. Planned actions include making the COGENT application independent/nondependent on the SCS tool while prototyping. Additionally, SCS results/output should be fed into the stand-alone COGENT application, and the output of COGENT should be compatible/usable by SCS. Since COGENT is a prototype, this risk has a lower severity.	4	A
S9	Software Deployment	3	E	Accept, Mitigate, Monitor	Related to S6, the COGENT application will need to be deployed in a manner that various users can interact with it. This will bring various software requirements (libraries, license for SCS, IPython/Conda), hardware (physical device, virtualization), operating systems (Windows, Ubuntu), and user interfaces (web-based browser, emulated notebook). Finding the best design will require additional effort/time, and thus, this is being listed as a risk. The risk is that additional resources to plan/manage these will cause delays in development.	1	C
S10	Plugin Misfiring	5	E	Mitigate, Avoid, Monitor	One challenge for developing a plugin manager with many plugins is they might have sequencing dependies. For example, a sequence might include data loading, transformation, usage, storage, and visualization (from storage). If these plugins fire out-of-order, the pipeline will fail.	5	A
S11	Evolving Third-Party Software	3	B	Accept, Mitigate, Monitor	A common issue with third-party interfaces and libraries is that they may change. Already, two of the third-party APIs have changed their format since the beginning of the project. This results in code updates/changes to address communication inconsistencies over the API. To minimize this, software has been developed with metamodals, object-orientation, and versioning so that only small changes to the interface or API call need to made.	3	B

B Appendix II - Requirements & Risk Register

B.1 CDF Positions & IDs

Table B.1: CDF Positions and IDs.

Position	ID
Documentation	DOC
Systems	SYS
Configuration	CNFG
Structure	STRUCT
Simulation	SIM
Attitude & Orbital Control System	AOCS
Propulsion	PROP
Mission Control	MISS
Communications	COMM
Ground Systems & Operations	GS & OPS
Data Handling System	DHS
Electric Power System	EPS/POW
Thermal Control System	TCS
Instruments	INST
Mechanisms	MECH
Programmatics	PROG
Risk Assessment	RISK
Cost Analysis	COST
Team Leader	TL

B.2 Software Requirements

Functional Requirements

No.	Name	CrossConcern	P	Description	Notes
R1	Users.Concurrent	Usability	H	The system shall support N concurrent user instances (multiple notebooks).	Tested: 4 simultaneous users in SCS and Google Colab
R1.1	Multi-User.Scoring	Efficiency	H	The system shall integrate the scoring results from multiple users.	Plugin
R1.2	Multi-User.Ranking	Efficiency	T	The system shall rank the combined results from user scoring.	Script: Ranking; Successfully ranked results from multiple users but not developed for COGENT. Transferred responsibility to SISW.
R2	Visualize.Pareto	Usability	H	The system shall plot all architecture variant FoMs with Pareto charts.	via SCS Discover, wandb
R3	Users.Interactive	Usability	H	The system shall allow for users to interact with charts and diagrams.	via Discover, wandb, Neo4J
R4	Doc.Design.Dec.	Explainability	H	The system shall allow users to input their design decisions.	Simulated via autonomous agents, data files
R4.3	Filter.Constraint	Flexibility	M	The system shall filter results based on user constraints.	via Discover, wandb
R5	Traceability.SEIM	Compliance	H	The system shall comply with the SEIM (e.g. as defined in CDP4 or ConOps).	via model definition
R6	User-Defined.Feat.	Extensibility	H	The system shall allow for user-defined rules, constraints, algorithms, features, and parameters.	via model definition, python scripts
R6.1	User-Created.Funct.	Extensibility	H	The system shall allow for user-created functionalities for scoring (e.g., addition vs multiplication in weights)	Plugin via python scripts, notebooks
R6.2	User-Defined.Filters	Extensibility	H	The system shall allow for user-defined filters (visualization and omiting/removing solutions).	via python scripts, Discover, wandb
R7.1	Multi-Quant.Scoring	Performance	M	The system shall automatically generate scores for architecture variants based on quantitative FoMs.	Simulated via autonomous agents, data files
R7.2	Multi-Quant.Ranking	Performance	T	The system shall automatically rank architecture variants based on quantitative FoM scores.	Script: Ranking; Transferred responsibility to SISW
R7.3	Multi-Qual.Scoring	Usability	M	The system shall allow designers to input their scores for qualitative attributes.	Simulated via autonomous agents, data files
R7.4	Multi-Qual.Ranking	Performance	T	The system shall automatically rank architecture variants based on qualitative FoM scores.	Script: Ranking; Transferred responsibility to SISW
R7.5	Multi-Arch.Scoring	Usability	M	The system shall allow designers to input their scores for architectural attributes.	Simulated via autonomous agents, data files
R7.6	Multi-Arch.Ranking	Performance	T	The system shall automatically rank architecture variants based on architectural FoM scores.	Script: Ranking; Transferred responsibility to SISW
R7.7	Multi-FoM.Scoring	Performance	M	The system shall automatically combine the scores from qualitative, quantitative, and architectural FoMs.	Simulated via autonomous agents, data files
R7.8	Multi-FoM.Ranking	Performance	T	The system shall automatically rank architecture variants based on quantiative, qualitative, and architectural FoM scores.	Script: Ranking; Transferred responsibility to SISW
R15.1	SYS.Likes	Funct. Req.	H	The system shall utilize user-inputted "likes."	Simulated via autonomous agents, data files
R15.2	SYS.Likes.Common	Explainability	L	The system shall identify commonalities in "likes."	Implicit, HITL
R15.3	SYS.Likes.Predict	Extensibility	L	The system shall predict which architectures will be "liked."	Plugin via sentiment analysis
R15.4	Multi-Obj.Optim.	Performance	L	The system shall utilize current methods for multi-objective optimization when generating scores.	Not relevant during the Design Space Exploration phase.
R16.2	Weights.Variable	Flexibility	L	The system shall allow for varying weights for subsystem KPIs.	Script: weighted_ranking; Successfully utilized adjustable weights for FoM but not developed for COGENT.
R17	Data.Store	Maintainability	H	The system shall be able to store all relevant (meta)data.	Plugins: AWS S3, Neo4J, ArangoDB, wandb
R17.1	Data.Store.Central	Reliability	H	The system shall be able to store all data in a centralized location.	Plugin: AWS S3
R17.2	Data.Store.Cloud	Portability	H	The system shall be able to store all data via cloud solutions.	Plugins: AWS S3, Neo4J, ArangoDB, wandb
R17.3	Data.Store.Local	Supportability	H	The system shall be able to store relevant data locally.	Storage via local machine, local terminal, discrete SCS instance.
R17.4	Data.Store.Level	Performance	H	The system shall be able to store metadata at the appropriate level.	Plugins: AWS S3 (JSON), Neo4J, ArangoDB
R17.5	Data.Store.GDB	Maintainability	H	The system shall support storage into a Graph Database.	Neo4J and ArangoDB were both implementated and available in COGENT.
R17.6	Data.Store.NoSQL	Maintainability	L	The system shall support storage into a NoSQL Database.	Amazon DynamoDB and MongoDB were successfully utilized but are not developed for the final implementation
R17.7	Data.Store.RDBMS	Maintainability	L	The system shall support storage into an RDBMS.	Amazon RedShift was successfully utilized but is not developed in the final implementation
R17.8	Data.Store.ExpTrack	Maintainability	H	The system shall support storage into a Machine Learning Experiment Tracking datastore.	Plugin
R17.9	Data.Store.ETL	Maintainability	L	The system shall support storage via an Extract, Transform, Load process.	Amazon Databrew Glue ETL was used for preprocessing and batching but not a core module
R17.10	Data.Store.Assets	Maintainability	H	The system shall support the storage of other assets (images, models, diagrams)	Scripts: s3_upload, s3_download

Software Requirements Continued

R18.1	Data.Format.CSV	Usability	H	The system shall support data in CSV format.	Plugin: CSV
R18.2	Data.Format.JSON	Usability	H	The system shall support data in JSON format.	Script: model_initialization
R18.3	Data.Format.ACEL	Usability	L	The system shall support data in ACEL format.	Environment: SCS
R18.4	Data.Format.CYPHER	Usability	M	The system shall support data in CYPHER format.	Plugin: Neo4J
R18.5	Data.Format.AQL	Usability	L	The system shall support data in AQL format.	Plugin: ArangoDB
R18.6	Data.Format.HDF5	Usability	M	The system shall support data in HDF5 format.	Plugin: HDF5
R19.1	Data.Xform.Vec	Performance	H	The system shall transform models in the JSON format into parameter vector form (structured).	Plugin (HDF5)
R19.2	Data.Xform.CSV	Performance	H	The system shall transform files in the JSON format to the CSV format.	Plugin: CSV-Discover
R19.3	Data.Xform.JSON	Performance	H	The system shall transform files in the CSV format to the JSON format.	Script: model_initialization
R19.4	Data.Xform.CYPHER	Performance	H	The system shall transform models in the JSON format to the CYPHER query language.	Plugin: Make-Cypher
R19.5	Data.Xform.AQL	Performance	M	The system shall transform models in the JSON format to the AQL query language.	Plugin: Make-AQL
R19.6	Data.Xform.ACEL	Performance	L	The system shall transform files in the ACEL DSL format to the JSON format.	SISW: "nice-to-have"
R20	Data.Merge	Usability	H	The system shall merge user-generated data and models.	Plugin: Merge-Models
R21	Plugin.Manager	Usability	H	The system shall support plugins.	Core: Host
R21.1	Plugin.Concurrent	Extensibility	H	The system shall support N concurrent plugins.	Tested: 20 simultaneous plugins
R21.2	Plugin.Timing	Reliability	H	The system shall support the ordered timing of plugins.	via Pluggy schedulers
R21.3	Plugin.User	Usability	H	The system shall support custom user plugins.	Plugin: Make-Correlation, Make-Sentiment
R21.4	Plugin.Deregister	Usability	H	The system shall support deregistering of plugins.	Cell: delete entry_points
R22.1	Single-Quant.Ranking	Performance	H	The system shall automatically rank architecture variants for a single user based on quantitative FoM scores.	Plugin: Weighted-Ranking
R22.2	Single-Qual.Ranking	Performance	H	The system shall automatically rank architecture variants for a single user based on qualitative FoM scores.	Plugin: Weighted-Ranking
R22.3	Single-Arch.Ranking	Performance	H	The system shall automatically rank architecture variants for a single user based on architectural FoM scores.	Plugin: Weighted-Ranking
R22.4	Single-FoM.Ranking	Performance	H	The system shall automatically rank architecture variants for a single user based on quantiative, qualitative, and architectural FoM scores.	Plugin: Weighted-Ranking

Quality Attributes (Non-Functional Requirements)

No.	Name	Concern	P	Description	Notes
NFR1	SYS.Conflict	Usability	M	The system shall support conflict resolution in decision-making sessions (align discussion).	Implicit, HITL
NFR2	Users.Expectations	Usability	H	The system shall support the balancing of multi-user expectations/needs.	Implicit, HITL, setting FoM in weighted_rank script
NFR3	SYS.Autonomy	Usability	M	The system shall be as autonomous as possible.	Plugin Manager
NFR4	SYS.Dependency	Modularity	H	The system shall be developed in an independent manner, not imposing dependencies on SCS.	Plugin Manager
NFR5	SYS/SW.License	Security	H	The system shall be developed based on approved licenses (e.g., open source)	GNU/MIT License
NFR6	Documentation	Explainability	H	The system shall be provided with a thorough ConOps and complete documentation.	SISW Project requirement
NFR7	"Pain" Input	Usability	M	The system shall address the major individual "pain" for subsystem teams.	HITL via quality attribute selection
NFR8	User.Errors	Usability	L	The system shall prevent user-based errors.	SISW: "nice-to-have"
NFR9	System.Install	Portability	M	The system shall be easy to install	Not all users are SW developers
NFR10	Multi-FoM.Classify	Explainability	M	The system shall differentiate between quantitative, qualitative, and architectural FoMs.	via (meta)model definition
NFR11	Security.Network	Security	L	The system shall meet the network security software requirements of Siemens SISW.	SISW: "nice-to-have"
NFR12	Security.Data	Security	L	The system shall securely store data.	AWS authentication
NFR13	Security.Database	Security	L	The system shall use Siemens SISW-based databases and database solutions.	SISW: "nice-to-have"

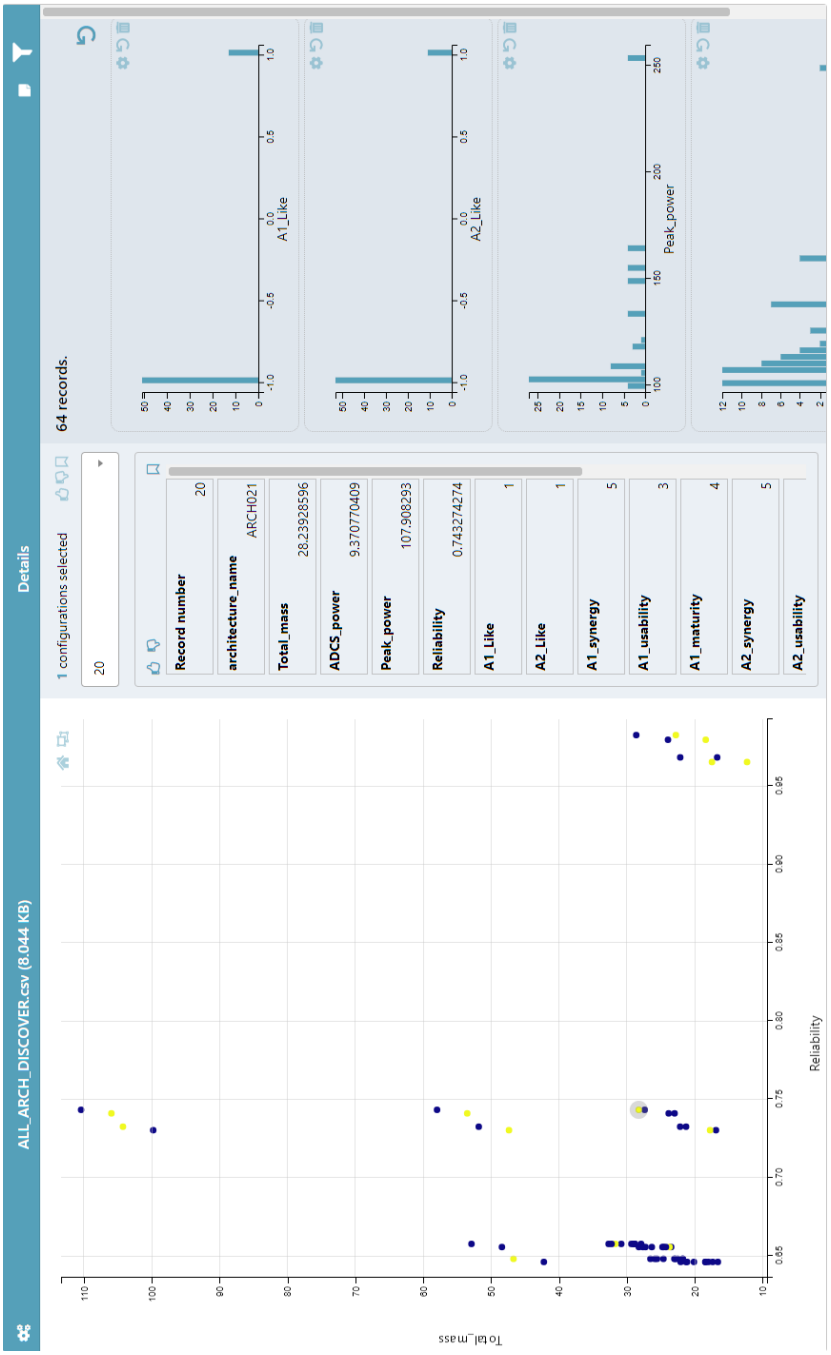

Figure B.1: Siemens Discover screen capture. All 64 architecture configurations have been plotted with Agent likes, quality attributes, and quantitative parameters values. ARCH021 is selected for having the highest simulated rank.

Fig. B.2 updates the previous Systems Functionality Description with explicit technologies used in the FireSat case study. Note that some modules were omitted to prevent cluttering.

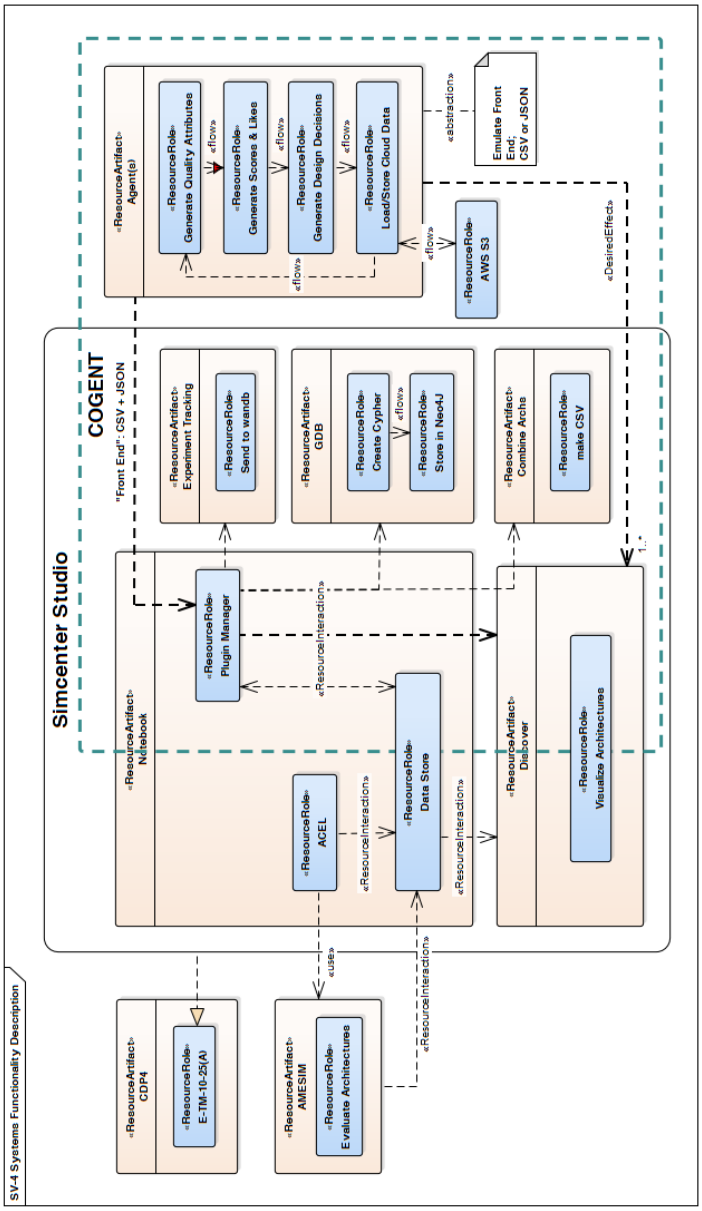

Figure B.2: FireSat Systems Functionality Description. The model demonstrates the high-level flow and resource interactions of the system with specific tooling. Quality attribute generation, scoring, and design decision notes are automatically generated by autonomous agents.

C Appendix III - Design & Technology Alternatives

C.1 High-Level Process Flow

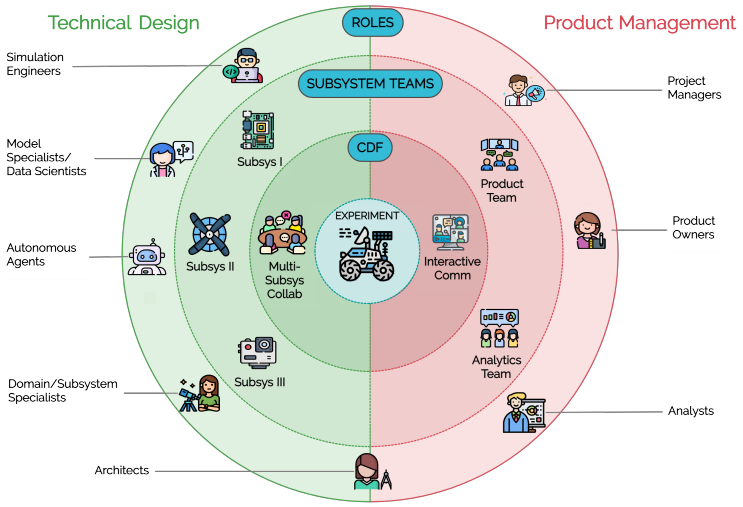

Figure C.1: General CDF Onion Diagram (high level of abstraction).

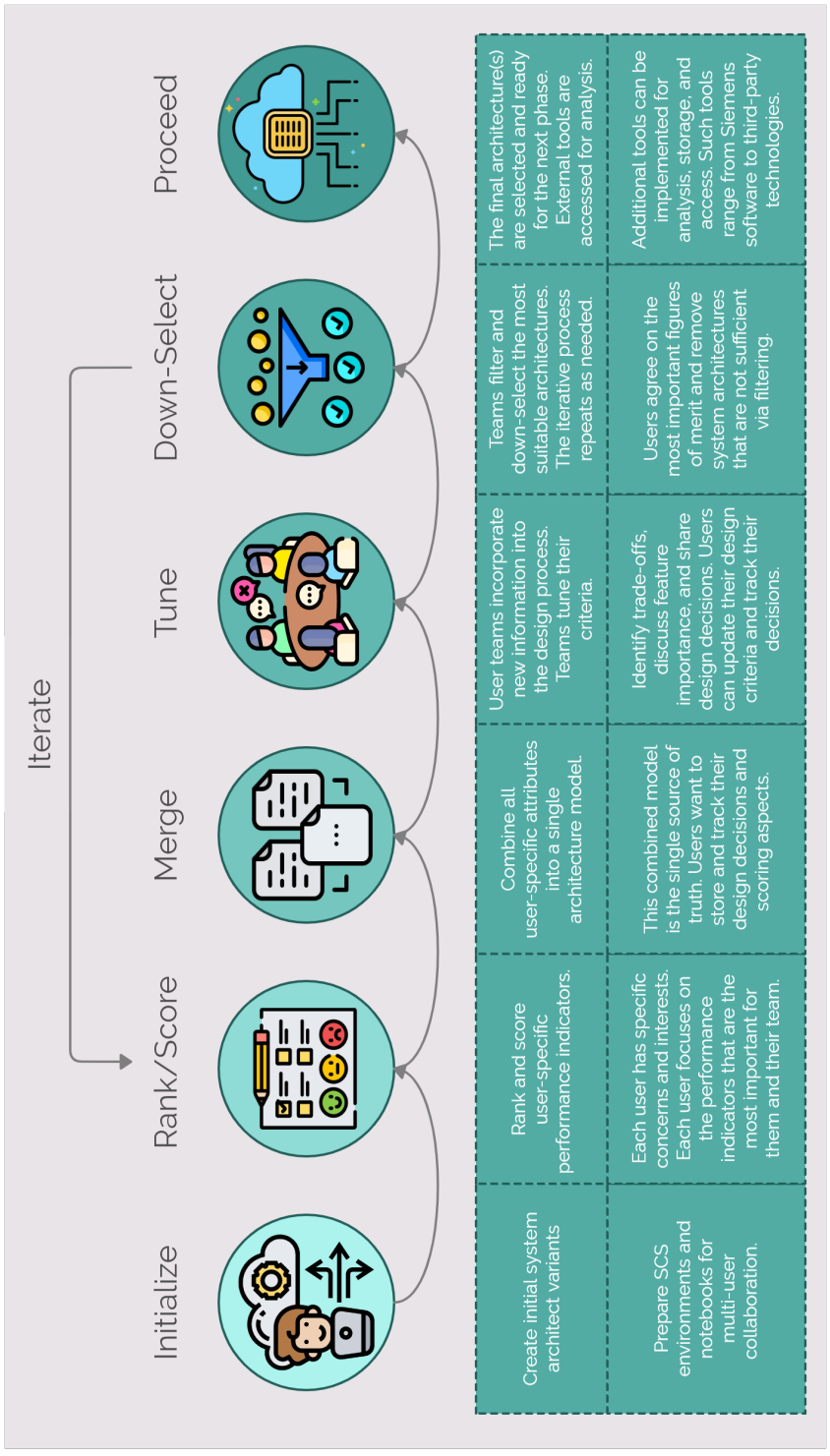

Figure C.2: Imagined Process Flow.

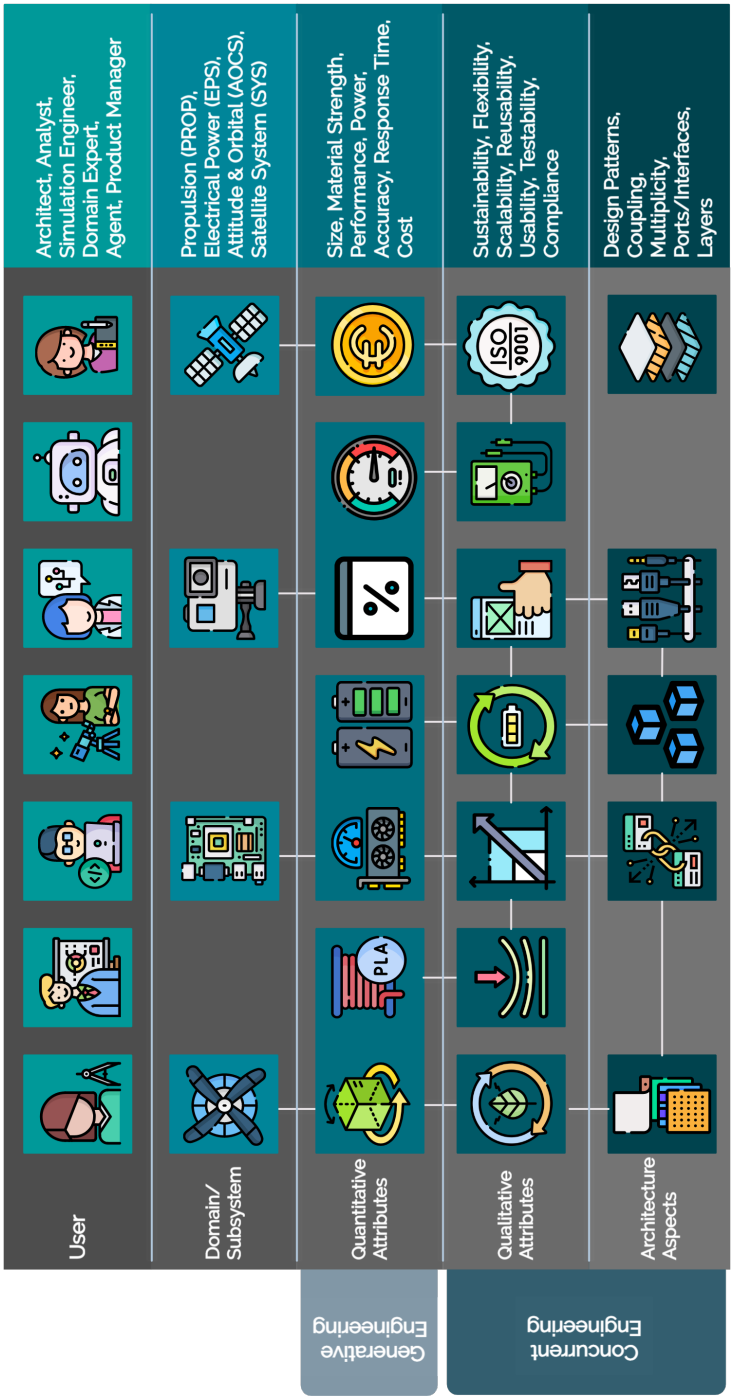

Figure C.3: Example performance indicators. Layers are focused on users, domain/subsystem, and attributes. Additionally, the domains in which GE and CE are needed for addressing quality attributes is provided. Attributes have intrinsic confounding and coupling.

C.2 Design Alternatives

An design alternative with COGENT external from SCS was considered. This would lead to a lower coupling and ability to use COGENT without SCS but at the cost of more interfaces and system resources. We decided against this alternative since SCS is meant to be a single-page application, keeping the User within the environment for completing tasks if possible.

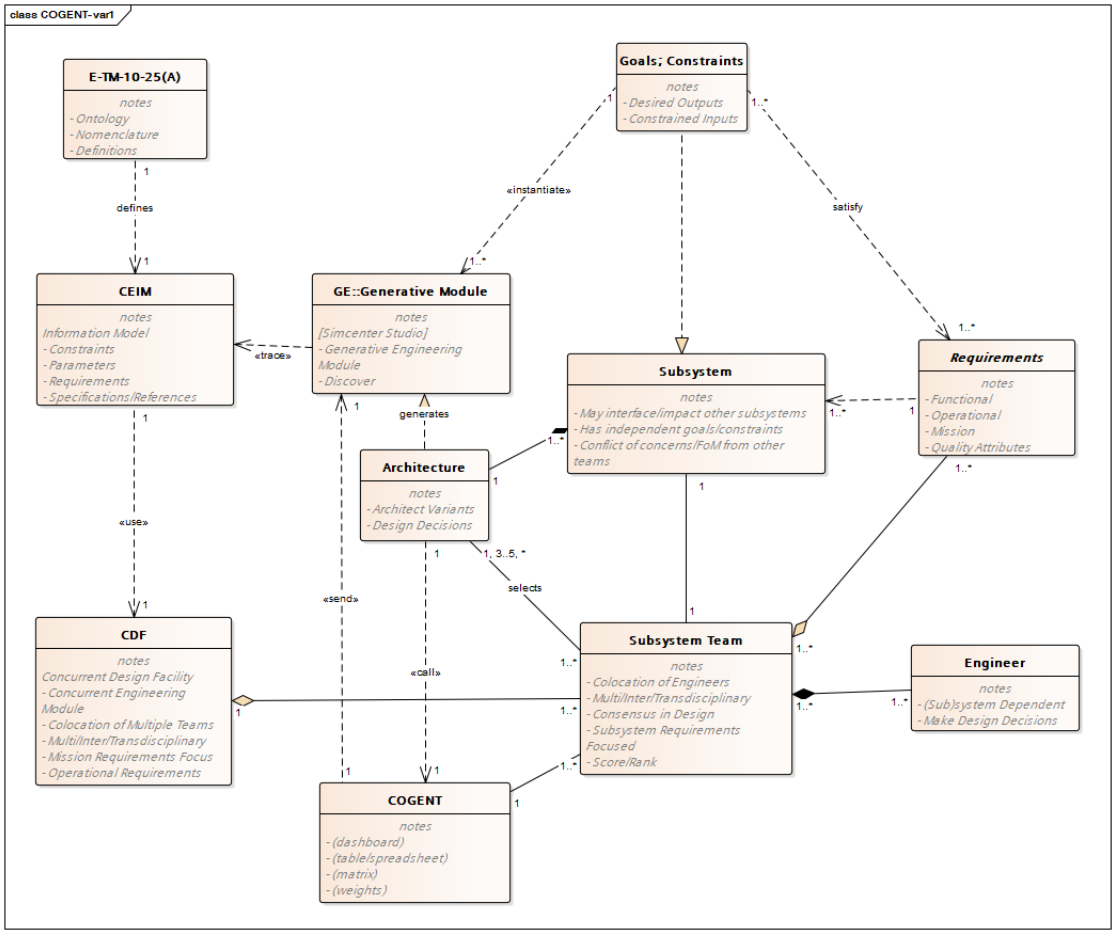

Figure C.4: COGENT Design Alternative (Variant 1) in which COGENT module is external from SCS.

Orchestrated Systems Event-Trace Diagram. Fig. C.5 demonstrates the interactions between the Orchestrator Actor and a number of User Actors. The Actors operated sequentially, with the architecture model merged after each Actor successfully completed their tasks. This model was abandoned for the concurrent, out-of-order operations model seen in Fig. C.6.

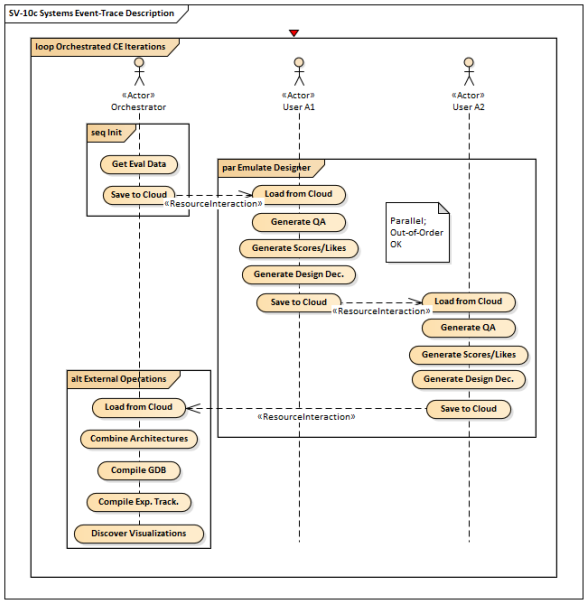

Figure C.5: Orchestrated Systems Event-Trace Diagram.

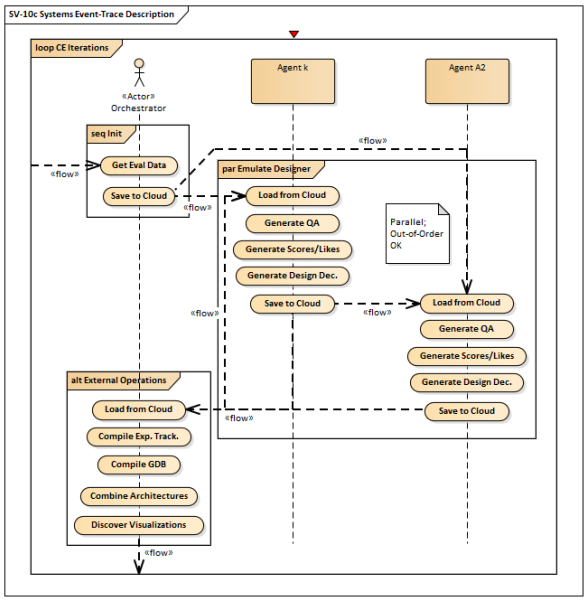

Figure C.6: Agent Systems Event-Trace Diagram.

Fig. C.6 demonstrates the interactions between the Orchestrator Actor and a number of autonomous Agents. The Agents are permitted to conduct some operations non-sequentially such as generating design decision, ranking, or saving the model to the Cloud after a single operation.

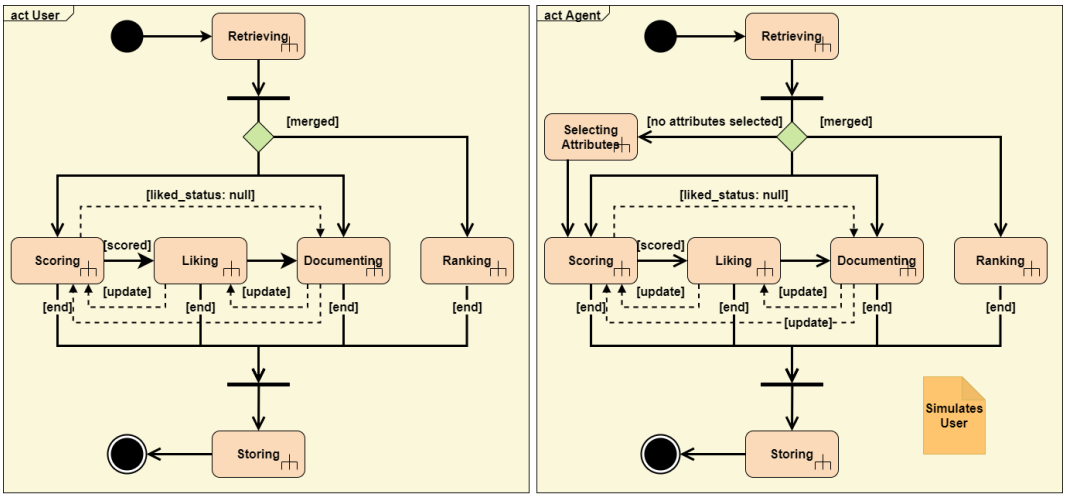

Figure C.7: Activity diagram demonstrating the sequence in which Users or Agents can score, like, rank, and add design decisions.

C.3 System Usage Scenarios

This section details six primary system usage scenarios and briefly describes scenarios for each use case. Additionally, an example of a misuse case is included describing the behavior that should be avoided. These scenarios are based on using COGENT in a CDF environment (extended from Fig. C.1, include Technical Design and Product Management sides). The data format is not fixed in the scenarios diagrams as assets may have different inputs and outputs. The conceptual position of COGENT is provided in the boxed-in area. The Architecture System level is a Conceptual Viewpoint. The Simcenter Studio level is a Logical Viewpoint. The Actors for Producer and Consumer are example Actors, as other Actors (e.g., Engineer, Manager) can easily replace the specified Actor type examples.

C.3.1 User Scenario 1: Single Producer/Consumer

In Scenario 1, a single designer is responsible for both the generation (creation) and analysis of architecture configurations. As a producer, this designer creates, edits, and stores architectures, architecture configurations, and models. The Producer-Actor should be from the Technical Design side.

Scenario: A single designer wants to independently generate and analyze an experimental concept.

C.3.2 User Scenario 2: Single Producer and Single Consumer

In Scenario 2, a single designer is responsible for the generation (creation) of architecture configurations. A single consumer (analyst) can analyze or view the architecture configura-

tions. The Producer-Actor should be from the Technical Design side. The Consumer-Actor should be from the Product Management side.

Scenario: A single designer wants to generate conceptual architectures together with a single analyst in collaboration.

C.3.3 User Scenario 3: Single Producer Team

In Scenario 3, the Producer-Consumer is from the Technical Design side. The Actors are (mostly) from the Technical Design side. While not shown, the process is iterative with the Producer team requiring multiple iterations of analysis, filtering, down-selecting, annotating, etc.

Scenario A: A single design team is exploring a potential experiment during Phase 0 (internally).

Scenario B: Multiple subsystem design teams are working concurrently to identify an optimal architecture configuration during Phase 0 (internally).

C.3.4 User Scenario 5: Single Consumer

In Scenario 5, there are no Producers (the project has been completed). However, a single consumer can analyze/interact with previous mission/experiment data.

Scenario: An analyst (e.g., Intern) is analyzing decision decisions from a data archive perspective on historical data.

C.3.5 User Scenario 6: Consumer Team

In Scenario 6, there are no Producers (the project has been completed). However, a Consumer (Analysis Team) can analyze/interact with previous mission/experiment data.

Scenario: An management team (e.g., Safety and Compliance) is analyzing decision decisions from a data archive perspective on historical data. In the event of a mission failure, the management team can potentially trace the cause of errors to the architecture (faulty components, improper interfacing, specific design decision, etc.).

C.3.6 Misuse Scenario Overview

Additionally, an example of a potential misuse scenario overview is shown in Fig. C.8. The issue shown is when two different teams are working independently on the same (set of) architectures.

Issues arise with:

- Data and/or models are not being constructed concurrently, coherently, or consistently
- Conflicting data locations, difficult to utilize results, conflict of access, conflict of data freshness, data might be invalid/inconsistent

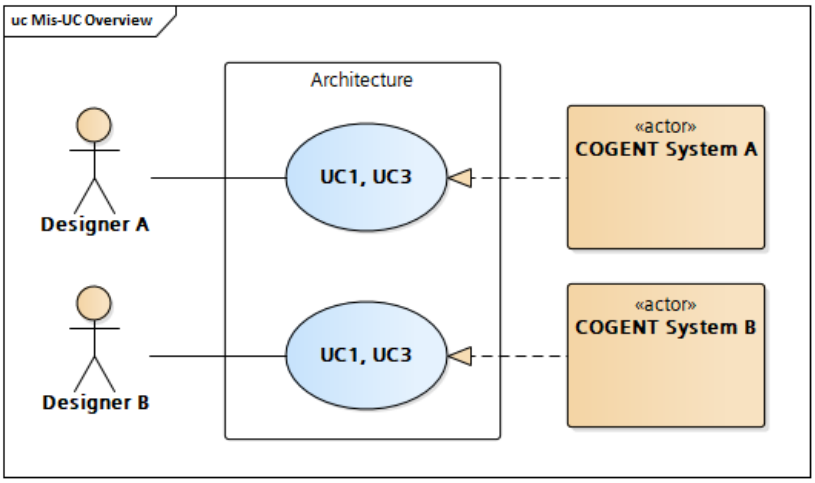

Figure C.8: Potential misuse case examples. These occur whenever a single designer or design team is interacting with generated architectures in parallel with other designers/teams (but referring to the same architectures).

C.4 Technology Alternatives

This section describes technology alternatives for architecture patterns, GDBs, experiment tracking, and data/object storage solutions.

C.4.1 Solution Architecture Analysis

In this section, a comparative analysis between between monolithic, microservice, and plugin (microkernel) architecture is detailed. The following aspects were considered:

- Agility: the ability to rapidly respond to a persistently changing environment
- Deployment: the ease of which the system is deployable
- Testability: the extent to which an objective and feasible test can be designed to determine whether a requirement is met
- Performance: the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage
- Scalability: the ability of the system to handle load increases without decreasing performance or increasing the load rapidly
- Development: the ease of which the system or components can be developed
- Customization: the ability for vendors and clients to create or include custom technologies and/or software modules, including databases, Python scripts, programming languages/wrappers, and third-party services
- Shareability: the permittable allowance of inter-user sharing of data, files, and models, in this case, especially configuration files. Note that sharing configuration files is not supported (as a practice) for microservice architectures.

Table C.1: Solution architectures pattern analysis.

	Monolithic	**Microservice**	**Plugin**
Agility	-	+	+
Testability	+	+	+
Performance	-	-	+
Scalability	-	+	-
Development	+	+	-
Customization	-	-	+
Shareability	-	-	+

C.4.2 Graph Database Comparison

This section compares three GDBs considered for implementation. Initially, several other GDB solutions were considered, but they were pruned. The three GDB below all allow for programming in Python, are ACID compliant [59], and strongly support concurrency and durability. Neo4J was selected for implementation for having the most advantages for COGENT's software development since Neo4J has the largest graph data community, the easiest to utilize APIs, extensive libraries, and direct support for graph data science and user-centric data analysis. Additionally, a Neo4J GDB is easily extendable and updateable, whereas ArangoDB requires unique key-value pairs for every relationship, node, and collection. In production, this means that the automatic generation of GDB in ArangoDB is infeasible unless the previously-stored GDB is deleted each time the user wants to add, remove, or change metadata in the stored GDB.

Table C.2: Comparison of Graph Database Management Systems (G-DBMS).

	Neo4J	**Amazon Neptune**	**ArangoDB**
DBMS Model	**GDB**	**GDB/RDF**	**GDB/Doc/K-V**
License	**Open-Source**	**Commercial**	**Open-Source**
Cloud Only	**No**	**Yes**	**No**
Server-side scripts	**Yes**	**No**	**Yes**
Schema	**Schema-free & Optional**	**Schema-free**	**Schema-free**
Sec. Indexes	**Yes**	**No**	**Yes**
API Access	**Bolt/RESTful HTTP**	**RDF/TinkerPop**	**HTTP/Gremlin**
Query Language	**Cypher/GraphQL**	**SPARQL**	**AQL/GraphQL**

C.4.3 Experiment Tracking Comparison

Table C.3: Comparison of Experiment Tracking & Management Systems.

	wandb	**neptune.ai**	**MLFlow**
Open Source	**No**	**No**	**Yes**
Lightweight	**Yes**	**Yes**	**Yes**
Data Versioning	**Yes**	**Limited**	**No**
Notebook Versioning	**Yes**	**Yes**	**No**
Model Versioning	**Yes**	**Limited**	**Limited**
Environment Versioning	**Limited**	**Limited**	**Limited**
Logging Artifacts	**Yes**	**Yes**	**Yes**
Run Grouping	**Yes**	**Yes**	**Limited**
View Sharing	**Yes**	**Yes**	**Limited**
Fetch Exp via API	**Yes**	**Yes**	**Yes**
Scale to 1M+ Runs	**Yes**	**Yes**	**No**

C.4.4 Cloud Object Storage Comparison

We considered four technologies for cloud-based object storage solutions, including Amazon AWS S3, IBM Cloud Object Storage, Azure Blob Storage, and Git. The AWS, IBM, and Azure solutions have similar specifications and performance values for durability, scalability, availability, data recovery, authorization (security), and costs. Of these, AWS is the easiest to integrate into a Notebook or Python environment with well-defined libraries, SDKs, and APIs. Comparing AWS and Git shows that both technologies have strong versioning capabilities. We decided to use AWS in our proof-of-concept implementation since it is the easiest to implement, extend, access, and monitor. Furthermore, AWS has compatible data warehousing, data transformation, and analytical technologies, whereas Git is simply a storage methodology.

C.5 RDBMS & GDB Metamodels/Schema

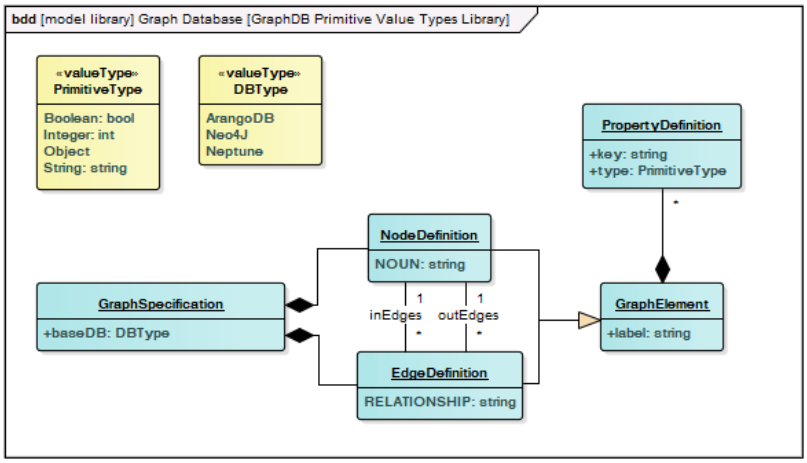

Figure C.9: Primitive Value Types model for the Graph Database implementation.

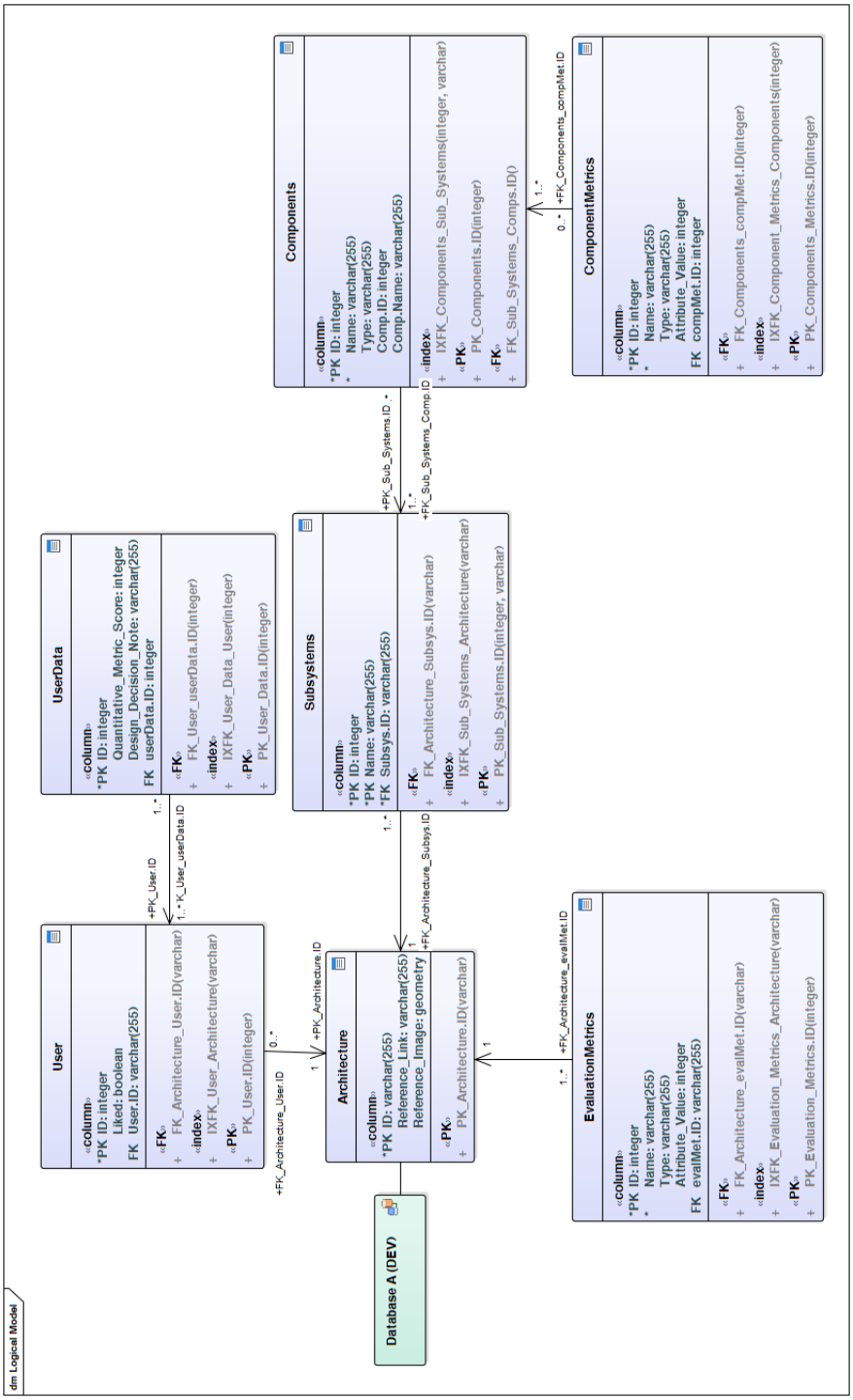

Figure C.10: Normalized Logical Model used for column-based RDBMS and NoSQL database design [7].

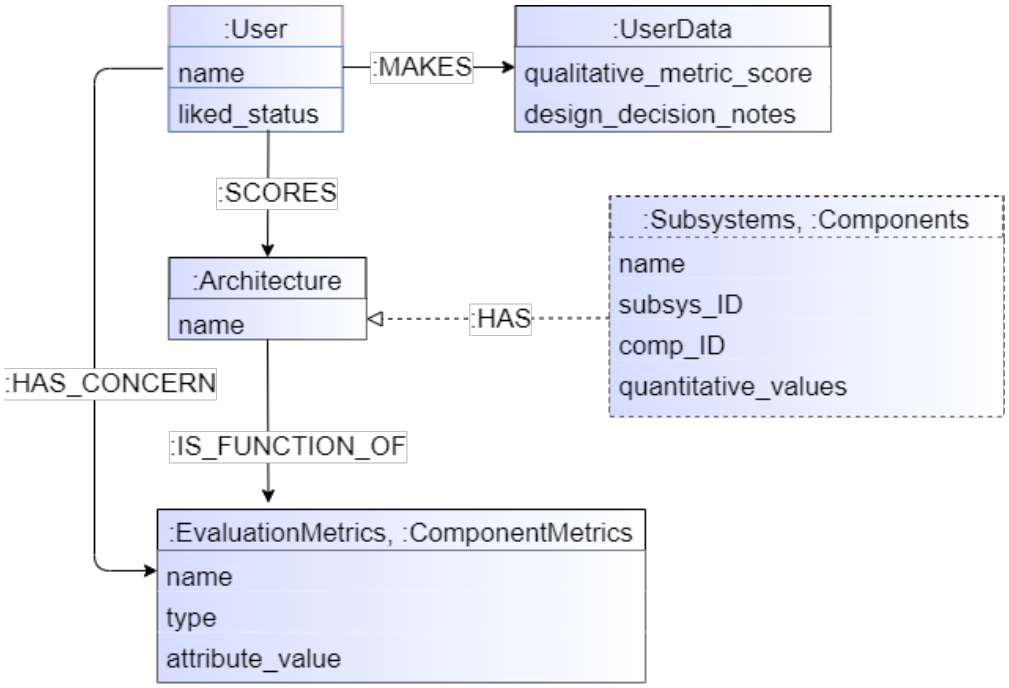

Figure C.11: Graph Database Schema based on Fig. C.10. Derived via RDBMS to GDB transformation steps in [8].

C.6 Additional Technology Output

C.6.1 Weights & Biases Cases

architecture	A1_Like	A2_Like	A3_Like	A1_usability	A2_usability	A3_usability	A1_synergy	A2_synergy	A3_synergy	A1_maturity	A2_maturity	A3_maturity	ADCS_power	Peak_power	total_mass	reliability
ARCH064	-1	-1	1	1	5	5	4	2	3	4	3	4	10.019	133.457	18.435	0.6461
ARCH064	-1	-1	1	1	5	5	4	2	3	4	3	4	10.019	133.457	18.435	0.6461
ARCH064	-1	-1	1	1	5	5	4	2	3	4	3	4	10.019	133.457	18.435	0.6461
ARCH064	1	-1	-1	3	2	1	5	2	4	4	3	3	10.019	133.457	18.435	0.6461
ARCH064	-1	-1	-1	5	3	4	2	3	1	2	2	5	10.019	133.457	18.435	0.6461
ARCH064	-1	-1	-1	1	4	2	2	2	1	4	5	5	10.019	133.457	18.435	0.6461

Figure C.12: Wandb. Six runs of the same architecture configuration are compared with different parameter values for attributes.

architecture	A1_Like	A2_Like	A1_synergy	A1_usability	A1_maturity	A2_synergy	A2_usability	A2_maturity	reliability	Panel	Battery	Fuel	total_mass	ADCS_power	Peak_power
ARCH026	1	-1	4	4	5	5	5	1	0.741	panel_GaAs_MJ	battery_NiCa	N2	53.512	10.249	102.553
ARCH025	-1	-1	4	3	1	4	5	2	0.7433	panel_Si	battery_NiCa	N2	57.957	10.252	102.611
ARCH024	1	-1	5	5	5	4	1	4	0.7303	panel_GaAs_MJ	battery_NiHy	Xe	17.736	9.366	107.906
ARCH023	-1	-1	1	5	1	2	5	1	0.7325	panel_Si	battery_NiHy	Xe	22.142	9.369	107.908
ARCH022	-1	-1	3	4	2	4	2	5	0.741	panel_GaAs_MJ	battery_NiCa	Xe	23.833	9.367	107.907
ARCH021	1	1	5	3	4	5	4	3	0.7433	panel_Si	battery_NiCa	Xe	28.239	9.371	107.908
ARCH020	-1	-1	2	2	3	3	2	2	0.7303	panel_GaAs_MJ	battery_NiHy	Xe	99.775	10.465	102.841
ARCH019	1	-1	4	4	5	2	3	5	0.7325	panel_Si	battery_NiHy	Xe	104.233	10.49	102.902

Figure C.13: Wandb. Eight runs of the different architecture configurations are compared against parameter values for attributes.

C.6.2 Neo4J Validation GDB

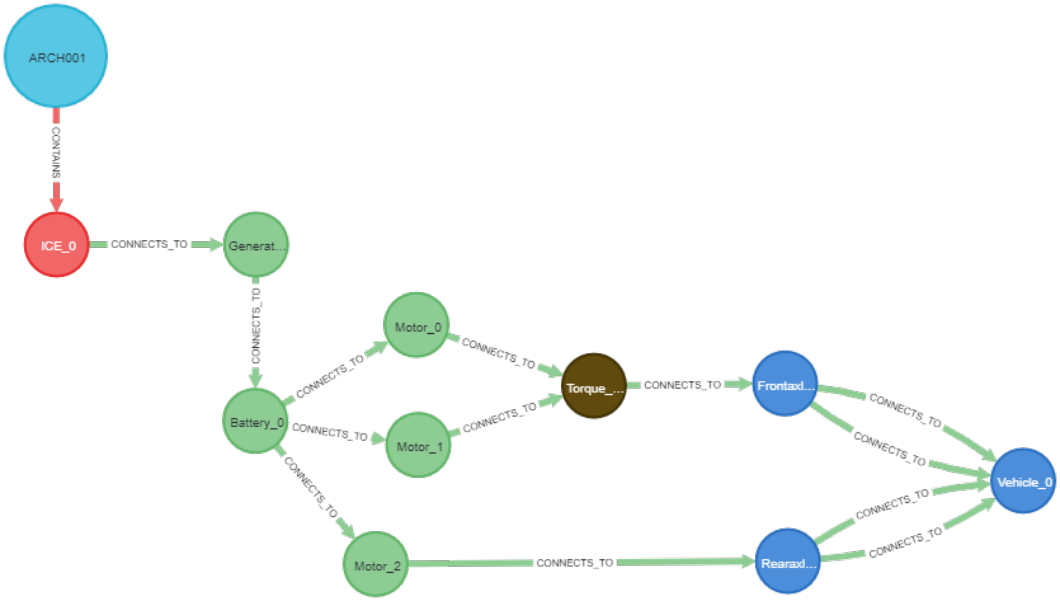

Figure C.14: Graph Database Interpretation of an Architecture Configuration generated from an ACEL Model.

C.6.3 Correlation Matrix

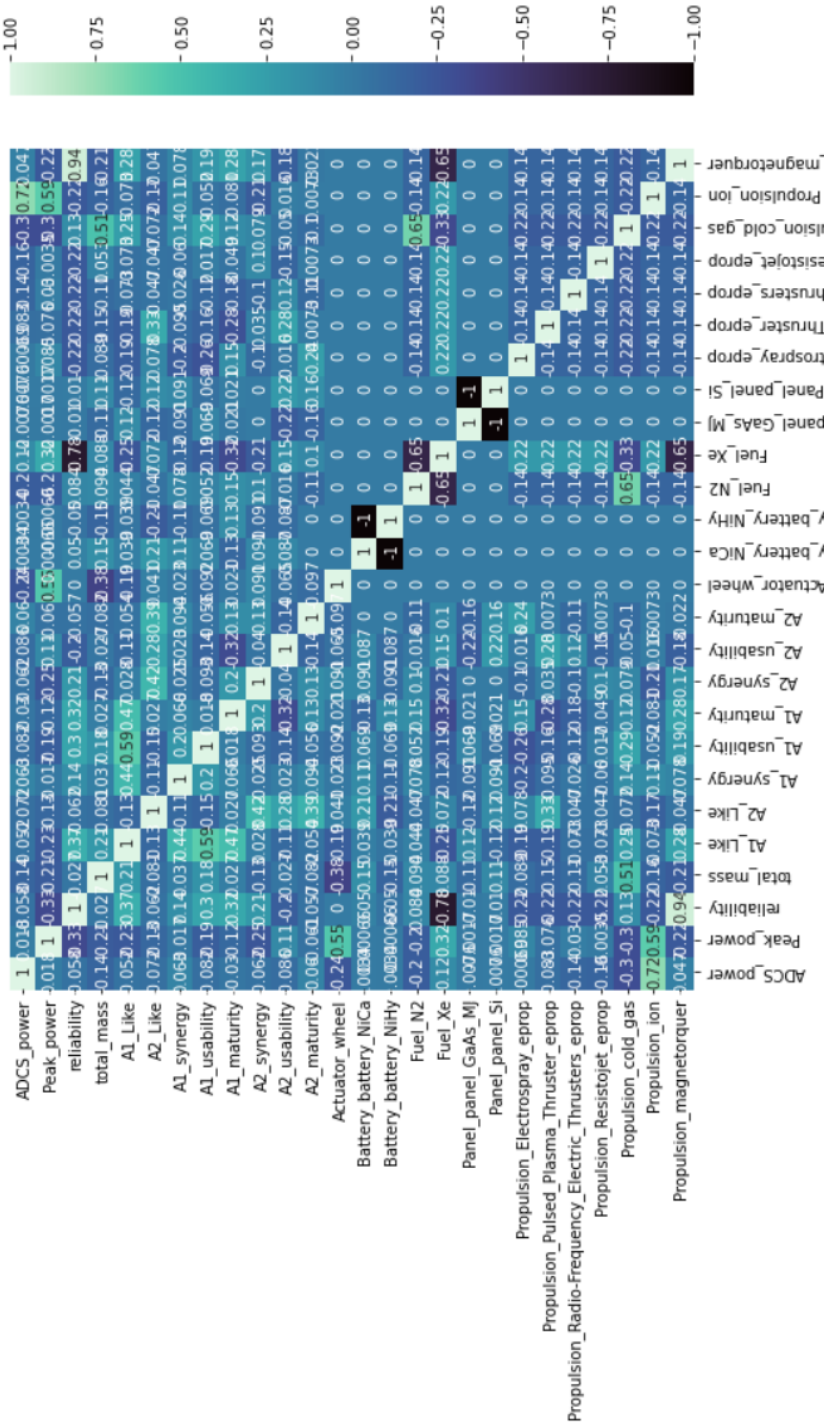

Figure C.15: Output Correlation Matrix. Values are between 1.0 and -1.0, which represent positively correlated and negatively correlated, respectively. The correlation demonstrates co-factor feature importance, e.g., using Xenon fuel has a highly negative impact on the Reliability.

C.7 Supplemental Design Information

This section contains supplemental design information and considerations when developing COGENT.

C.7.1 Concept of Operations

A Concept of Operations (ConOps) is a required deliverable to Siemens SISW and a major aspect of the project. A brief definition is provided below:

"Developed early in Pre-Phase A by the technical team, describes the overall high-level concept of how the system will be used to meet stakeholder expectations, usually in a time-sequenced manner. It describes the system from an operational perspective and helps facilitate an understanding of the system's goals. It stimulates the development of the requirements and architecture related to the user elements of the system. It serves as the basis for subsequent definition documents and provides the foundation for the long-range operational planning activities [14]." (NASA SE Handbook 2017)

C.7.2 Limitations in Pareto-optimal Diagrams

Pareto-optimal results are often displayed in two-dimensional diagrams. These diagrams are typically ideal for comparing two values when making trade-off decisions. A challenge arises whenever multiple (>2) subsystem teams have different views on what the most important comparable values are. Consider a satellite project containing four teams: the EPS team may be the most concerned with *energy consumption*, the AOCS team might be concerned with *downlink time*, the TCS team is concerned with *heat dissipation*, whereas management is concerned with *cost*. Plotting two of the values simultaneously is trivial but more than three becomes increasingly difficult (to the point of impossible).

Equally important is that the best architecture may not lie on the Pareto-optimal chart, since the "best" architecture may contain a balance of attributes that are non-optimal in any single FoM (but instead, a trade-off or balance between FoMs). Due to the limited amount of time designers can spend during DSE, the best architectures to meet mission objectives may never be selected. Since GE can derive (more than) thousands of architectures, how can we be sure that designers are able to identify truly suitable architectures within a finite time?

Multi-Objective Optimization (MOO) scoring techniques (Swarm algorithms, Linear Programming) are a possible alternative for ensuring that the best architecture is available for designers to rank/score. Furthermore, hard and soft constraints can be used as a filtering technique (e.g., the cost must never exceed X, or fuel consumption must never exceed Y). Additionally, machine learning can be implemented to derive the *feature importance* using Logistic Regression or Random Forests. This approach only works whenever a sufficient amount of (meta)data is available but would reduce the required designer ranking/scoring responsibilities.

C.7.3 Deriving FoM Priorities from Mission Requirements

The priority or importance of an FoM depends on the mission objectives and requirements. Mission requirements can vary greatly depending on mission objectives, environment, and constraints. For example, a communications satellite being developed to operate in LEO may have a priority in cost (deployment, operational) over robustness (life-cycle). In contrast, a satellite designed for deep-space exploration or orbiting another planet may prioritize life-cycle over cost, as the increased distance from Earth and increase life-cycle may justify neglecting the maximum cost.

C.7.4 Transient Definitions and Statuses in Attributes

In typical data analysis and analytics, attributes can be divided into qualitative and quantitative categories. Additionally, attributes can be classified as subjective or objective. The general assumption is that quantitative values like cost are objective, as the value does not change depending on a designer's perspective; the value is fixed. Similarly, qualitative attributes such as *usability* are considered subjective, as the quality of the qualitative scoring highly depends on the individual designer's perspective, experience, or biases.

From an architectural perspective, there are qualitative attributes that can be assigned as objective, such that a subjective attribute can transition into an objective attribute. For example, a component can be prescribed as *reusable*, i.e., the component has been developed in a manner that reusable. Initially, this assignment is subjective as it is *possible* that the component has reusable features (physically reusable, reusable software design patterns, etc.). If this same component has actually been reused in several architectures, then this becomes an objective assignment as there is evidence that the component is reusable. This distinction is nuanced but important when considering how qualitative attributes and their scoring impact both design decisions and interpretation of ontology/nomenclature.

Furthermore, a quality attribute or FoM may contain discrepancies in its definition that should be handled carefully. The quality attribute *"sustainable"* may have different meanings between subsystem groups or departments working on the same product. Does sustainability always mean that the product or architecture promotes ecological aspects, or, does sustainability mean that the product is able to support itself throughout its life-cycle? These definitions must be handled carefully and without ambiguity, with a formal definition of both what is and what is not contained in the keyword referenced (in the ontology).

PO Box 513
5600 MB Eindhoven
The Netherlands
tue.nl

PDEng SOFTWARE TECHNOLOGY