



**Technical University of Crete**  
**School of Electrical and Computer Engineering**

**Argumentation-Based  
Decision Support System for Systems  
Deployment:  
Case Study in the Ministry of Digital Governance**

**Diploma Thesis**

**Author:** Ioannis Michalakis

**Supervisor:** Michail G. Lagoudakis, Professor, School of ECE, TUC

**Member:** Antonios Deligiannakis, Professor, School of ECE, TUC

**Member:** Nikolaos Spanoudakis, Assistant Professor, Department of EE, HMU

Chania, July 2025



## **Πολυτεχνείο Κρήτης**

**Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών**

# **Ένα Βασισμένο στην Επιχειρηματολογία Σύστημα Υποστήριξης Αποφάσεων για την Εγκατάσταση Συστημάτων: Μελέτη Περίπτωσης στο Υπουργείο Ψηφιακής Διακυβέρνησης**

## **Διπλωματική Εργασία**

**Συγγραφέας:** Ιωάννης Μιχαλάκης

**Επιβλέπων:** Μιχαήλ Γ. Λαγουδάκης, Καθηγητής, Σχολή ΗΜΜΥ, ΠΚ

**Μέλος:** Αντώνιος Δεληγιαννάκης, Καθηγητής, Σχολή ΗΜΜΥ, ΠΚ

**Μέλος:** Νικόλαος Σπανουδάκης, Επίκουρος Καθηγητής, Τμήμα ΗΜ, ΕΛΜΕΠΑ

Χανιά, Ιούλιος 2025

## Abstract

The increasing complexity of software infrastructure management, particularly in public sector cloud deployment, necessitates intelligent decision-support mechanisms. Manual ways for decision making are very complex, results are inconsistent, and most of the times they waste resources. Automated solutions are crucial for optimizing key factors, including urgency, determining how quickly an application should be deployed along with infrastructure type, deployment location, resource allocation, and scalability. This diploma thesis introduces a new decision support system that uses argumentation to automate deployment choices for the Ministry of Digital Governance. The system uses different frameworks to balance competing needs, rules, and policies, which makes decisions both justified and easy to understand. The proposed system leverages Gorgias Cloud for reasoning, based on preferences, and Prolog to formally represent arguments. The system also uses Raison AI, which allows the creation of a complex decision policy, using a no-code symbolic interface to produce Prolog files. The backend was built using Spring Boot, a Java-based framework, to manage the entire process, creating both a decision strategy and YAML configuration files that help automate deployment. The proposed system enables users to define key parameters, such as infrastructure type (on-premise vs. cloud), scalability, resource requirements, location constraints, and urgency levels. Based on these inputs, the system evaluates trade-offs and generates optimised, explainable deployment strategies. A key contribution of this work is its ability to bridge the gap between technical decision-makers and policy stakeholders, providing a transparent and structured decision rationale. The results show, that using argumentation-based automation, can really improve the way government organizations adopt cloud technologies. In the future, the system could be expanded to integrate with DevOps pipelines, learn rules on its own, and even be used for things beyond just managing cloud infrastructures.

**Keywords:** Argumentation Theory, Cloud Deployment Automation, Public Sector IT, AI Reasoning

## Περίληψη

Η διαχείριση υποδομών λογισμικού καθίσταται ολοένα και πιο περίπλοκη, ιδιαίτερα στον δημόσιο τομέα, ως προς τις τεχνολογίες cloud που σχετίζονται με την ανάπτυξη πληροφοριακών συστημάτων. Οι χειροκίνητες μέθοδοι λήψης αποφάσεων είναι αρκετά πολύπλοκες, δεν προσφέρουν συνέπεια και συχνά οδηγούν σε σπατάλη πόρων. Για την επίλυση αυτού του προβλήματος, είναι αναγκαία η χρήση αυτοματοποιημένων εργαλείων που υποστηρίζουν κρίσιμες αποφάσεις, όπως το πόσο γρήγορα θα αναπτυχθεί μία εφαρμογή, το είδος της υποδομής που θα χρησιμοποιηθεί, ο τόπος υλοποίησης, οι απαιτούμενοι πόροι και η δυνατότητα κλιμάκωσης του συστήματος. Η παρούσα διπλωματική εργασία παρουσιάζει ένα νέο σύστημα υποστήριξης αποφάσεων, το οποίο αξιοποιεί την τεχνική της επιχειρηματολογίας για την αυτοματοποίηση επιλογών ανάπτυξης στο Υπουργείο Ψηφιακής Διακυβέρνησης. Το σύστημα βασίζεται σε διαφορετικά πλαίσια, ώστε να εξισορροπεί αντικρουόμενες ανάγκες, κανόνες και πολιτικές, καθιστώντας τις αποφάσεις τεκμηριωμένες και εύκολα κατανοητές. Συνδυάζει διάφορα εργαλεία, όπως το Gorgias Cloud για λογική βασισμένη σε προτιμήσεις και την γλώσσα Prolog για τυπική αναπαράσταση λογικής. Για την δημιουργία της λογικής με χρήση επιχειρημάτων χρησιμοποιήθηκε το Raison AI, το οποίο επιτρέπει την παραγωγή αρχείων σε γλώσσα Prolog, χωρίς γνώση προγραμματισμού, επικεντρώνοντας έτσι στην πολυπλοκότητα των αποφάσεων. Το backend έχει υλοποιηθεί με το Spring Boot Java-based framework, για να διαχειρίζεται ολόκληρη την web εφαρμογή, δημιουργώντας τόσο την ροή εργασίας, όσο και αρχεία παραμετροποίησης YAML για την αυτοματοποίηση της ανάπτυξης. Οι χρήστες μπορούν να καθορίσουν σημαντικούς παράγοντες, όπως αν θα χρησιμοποιηθεί υποδομή cloud ή on-premise, ποιά θα είναι αυτή η υποδομή, το εύρος της απαιτούμενης κλιμάκωσης, οι αναγκαίοι πόροι και τον βαθμό επείγοντος της ανάπτυξης. Το σύστημα αναλύει όλους τους πιθανούς συνδυασμούς και προτείνει μία αποδοτικότερη στρατηγική, εξηγώντας παράλληλα με σαφήνεια τις αιτίες των επιλογών του. Μία βασική συνεισφορά του προτεινόμενου συστήματος είναι η ικανότητα να γεφυρώσει το χάσμα μεταξύ των τεχνικών υπευθύνων λήψης αποφάσεων και των ενδιαφερόμενων στρατηγικών εταίρων, παρέχοντας μια διαφανή και δομημένη λογική λήψης αποφάσεων. Τα αποτελέσματα δείχνουν ότι η αυτοματοποίηση με βάση την επιχειρηματολογία, μπορεί να βελτιώσει ουσιαστικά τον τρόπο με τον οποίο οι οργανισμοί δημόσιας διοίκησης υιοθετούν τεχνολογίες cloud. Στο μέλλον, το σύστημα θα μπορούσε να επεκταθεί, ώστε να ενσωματωθεί σε DevOps pipelines, να μαθαίνει κανόνες αυτόματα και να βρίσκει εφαρμογή σε περιοχές πέραν της διαχείρισης υποδομών cloud.

**Λέξεις-κλειδιά:** Επιχειρηματολογία, Αυτοματοποίηση Ανάπτυξης Συστημάτων στο Νέφος, Πληροφοριακά Συστήματα Δημόσιου Τομέα, Συλλογιστική Τεχνητής Νοημοσύνης

# Contents

<b>1</b>	<b>Introduction</b>	<b>10</b>
1.1	Concept	10
1.2	Thesis Contribution	11
1.3	Thesis Outline	12
<b>2</b>	<b>Literature Review</b>	<b>14</b>
2.1	Argumentation	14
2.2	Preference-Based Argumentation	15
2.3	Computational Tools for Argumentation	16
2.3.1	Prolog as a Foundation for Computational Argumentation	16
2.3.2	Gorgias Cloud as a Tool for Argumentation-Based Reasoning	17
2.3.3	Raison: No-Code Symbolic Argumentation-Reasoning Platform	18
2.4	Technologies for Web Application and Integration	20
2.4.1	Technical Background: HTTP Methods and JSON Format	20
2.4.2	Spring Boot Framework for Web Service Development	22
2.5	Dynamic Web Content Thymeleaf Template Engine	23
2.6	Deployment Technologies: Docker and Git	24
<b>3</b>	<b>Methodology and System Design</b>	<b>29</b>
3.1	System Architecture Overview	30
3.2	System Workflow and Behavior	31
3.3	User Interface Design	41
3.3.1	User-Guided Configuration through Decision Forms	41
3.3.2	User Interface Views	44
3.4	Back-end Logic and Processing	48
3.4.1	UI Data Submission	48
3.4.2	Dispatching the Gorgias Query via a Swagger-Generated Client	49
3.4.3	Post-processing of Gorgias Results	50
3.5	Argumentation Engine Design	52
<b>4</b>	<b>Implementation</b>	<b>57</b>
4.1	Technology Stack and Tools	57
4.2	Development Environment and Project Setup	60
4.2.1	VS Code and Spring Boot Setup	60
4.2.2	Dependency Management: pom.xml	60
4.2.3	Integration of Gorgias Cloud API via Swagger Codegen	61
4.2.4	Version Control with Git	63
4.2.5	Containerization with Docker	63
4.3	Automated Configuration File Generation	64

4.3.1	How the Final Deployment Strategy Is Determined . . . . .	64
4.3.2	Official Flowchart Connection System Output . . . . .	66
4.3.3	Sample YAML Template . . . . .	67
<b>5</b>	<b>Evaluation And Results</b>	<b>68</b>
5.1	System Usability Scale (SUS) Questionnaire . . . . .	68
5.2	Analysis of Results Using Boxplot Diagrams . . . . .	69
<b>6</b>	<b>Conclusions and Future Work</b>	<b>72</b>
6.1	Future Work and Enhancements . . . . .	72
6.2	Final Thoughts . . . . .	73

# List of Figures

2.1	Simple argumentation graph for cloud migration decision. Solid arrows indicate support, dashed arrows indicate attacks. . . . .	15
2.2	Visual rule definition in <i>raison</i> : Scenarios mapped to options. . . . .	19
2.3	Conflict resolution interface: Choosing the preferred scenario to ensure consistent reasoning. . . . .	19
2.4	Querying a rule-based decision model: Given a set of input facts, the system evaluates the corresponding rules and returns the derived conclusion. . . . .	20
2.5	Basic REST API Communication Flow . . . . .	22
2.6	Layered architecture of a Spring Boot web application, showing main components, user flow, database, and integration with external APIs, such as Gorgias Cloud. . . . .	23
3.1	High-level system flow: User input is transformed into Prolog facts, processed via the Gorgias Cloud Platform, and the output is interpreted into human-readable decisions and optionally exported as a YAML file. . . . .	31
3.2	User Interface for Urgency Parameter Selection — aligned with Ministry rules for urgency classification . . . . .	32
3.3	User Interface output showing the recommended urgency classification . . . . .	37
3.4	Overview of the user workflow for case-based decision support. . . . .	38
3.5	Interface showing applying final selections . . . . .	38
3.6	Recommended Deployment Strategy: Azure Functions (Premium) . . . . .	39
3.7	Technical Reasoning Behind the Deployment Decision . . . . .	39
3.8	Urgency step – collecting deployment urgency and organizational context. . . . .	44
3.9	Infrastructure preferences – selecting appropriate infrastructure model types (e.g., SaaS, PaaS, IaaS, Serverless). . . . .	44
3.10	Location selection – defining regional and compliance requirements for on-premises or public cloud deployment. . . . .	45
3.11	Resource configuration – prioritizing compute, memory, storage, and latency needs. . . . .	45
3.12	Performance parameters – defining expected traffic volume, spikes, and scalability behavior. . . . .	46
3.13	Final decision interface – the user reviews the collected inputs and selects between appropriate decisions . . . . .	46
3.14	Final selection submission – the user applies the selected deployment configuration. This action triggers the backend processing engine to finalize the decision logic and prepare for strategy execution. . . . .	47
3.15	System’s final decision output – this view presents the recommended deployment strategy derived from the user’s structured inputs, supported by reasoning and explanation generated by the argumentation engine. . . . .	47

3.16	Scenario modeling using Raison AI . . . . .	54
3.17	Conflict resolution interface (8/8 resolved) . . . . .	54
3.18	Project execution to test logic . . . . .	54
3.19	Scenarios for infrastructure . . . . .	55
3.20	Decision matrix for scalability . . . . .	55
3.21	Scenario and outcome mapping for location policy . . . . .	55
3.22	Modular Prolog rule sets hosted and executed in Gorgias Cloud . . . . .	56
4.1	Enhanced implementation architecture with RESTful requests and responses, internal Spring Boot logic, and integration with external reasoning and rule tools. . . . .	59
4.2	Azure service selection flowchart used by the Ministry of Digital Governance	65
5.1	Distribution of participant responses for each System Usability Scale (SUS) question, as shown in boxplot diagrams. Each boxplot displays the median (red horizontal line), the interquartile range (IQR; box), and potential outliers (points beyond whiskers). The box represents the middle 50% of responses, while the whiskers indicate the full range, excluding outliers. This visualization enables rapid identification of consensus or disagreement among participants for each question. Higher medians and tighter boxes indicate greater consensus and satisfaction. . . . .	69
5.2	Distribution of calculated SUS scores across all participants. . . . .	71



# List of Tables

2.1	Common HTTP Methods in RESTful APIs . . . . .	21
2.2	Common HTTP status codes for RESTful API responses . . . . .	21
3.1	Responsibilities of system components . . . . .	48

# Chapter 1

## Introduction

### 1.1 Concept

Decisions on cloud deployment, in contemporary public sector environments, have grown increasingly difficult and call for organisations to strike a balance, between technical needs, operational restrictions, and regulatory compliance. In this context, argumentation-based reasoning [1], a methodical approach to logically infer conflicting information, offers a strong technique for automating and justifying deployment decisions. This thesis, uses computational argumentation, to enable the automated choice of the best cloud deployment strategy for hybrid government clouds, containing public sector data systems. This hybrid infrastructure, allows a dynamic and policy-driven approach to infrastructure provisioning, by combining on-site data centres run under General Secretariat control, with public cloud services, including Microsoft Azure. Through the use of argumentation, the system guarantees that deployment decisions are open, understandable, and consistent with government policies, while so optimising resource use, economy, and security issues.

The project started with a critical collaborative phase with the **Ministry of Digital Governance**, more especially the Information Systems Department of the General Secretariat of Information Systems, to guarantee the system addressed real-world requirements. By means of seminars and dialogues with ministry stakeholders, we discovered important decision-making elements, affecting public sector IT implementations, such as scalability, security, compliance, budgetary restrictions, and operational urgency. These pragmatic issues were then converted into a formal argumentative structure, whereby the definition of formal rules and the creation of a preference hierarchy, could help to settle disputes. This close cooperation guaranteed, that the design of the system was based on real operational needs, and regulatory reality, hence the produced solution is strong and pertinent for application, in the Greek public sector.

The “Deployment Choice Automation Using Argumentation” [2] system enables users to define various parameters across several decision domains. These include:

1. **Urgency Details:**

Users input critical information, such as the initiating organisation, contract status and planned start date. These details are pivotal in establishing the urgency level: Normal, High, or Urgent. It helps to prioritise tasks, and set the deployment timeline efficiently.

2. **Infrastructure Requirements:**

Users select the type of infrastructure, that best suits their application’s needs. Whether the goal is to run custom applications, consume ready-made software, or automate

specific tasks, the system considers the level of control required over the infrastructure (full, limited, or none).

**3. Deployment Location:**

The system collects data on sensitivity and connectivity requirements, enabling it to decide whether the solution should be deployed in an on-premise data center, or in a public cloud environment, like Azure. Factors, such as data sensitivity (e.g., personal, critical, or regulated data) and connectivity to legacy systems, are taken into account.

**4. Resource Requirements:**

In this case, input is about the computational resources, such as CPU, memory, and storage. This step ensures that the chosen solution will meet performance demands under various operational conditions.

**5. Scalability and Performance Requirements:**

Anticipated load conditions, expected peak times, and scalability needs are specified to ensure that the deployment can adapt dynamically to varying demand levels.

Once the system collects the required parameters, after each case submission, it integrates with Gorgias Cloud, where the argumentation process occurs [3][1]. The decision logic is modelled using Prolog rules. These rules are generated and maintained using the Raison AI platform. They formalise competent factors, that are crucial for making the optimised choice for each case.

Initially, it determines the most suitable **Infrastructure** cloud service, selecting from Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), or Serverless computing, based on the specific requirements of each public sector deployment scenario.

Then, it determines the Deployment **Location**, and recommends either on-premises or public cloud options.

Another case checks the **Urgency** timeline. It categorises the requests into three levels: normal, urgent, or high-priority. Urgent priority requests are allocated resources before others. Normal ones wait their turn.

**Scalability** is another factor. The system chooses between auto-scaling, which changes resources as needed, or fixed resources, which always stay the same.

In **Resources** case, it picks the right hardware, like high-memory servers for data-heavy apps. Before suggesting the final decision, the system shows up a final step page, where the users can choose for each case the suggested decisions that match their needs.

At the end, the system creates deployment configuration files, named YAML (stands for Yet Another Markup Language), ready for DevOps teams to use. This saves time and reduces errors. By integrating computational argumentation into cloud deployment planning, this system provides a structured, transparent, and user-centric solution to the complex challenge of cloud infrastructure selection for the public sector.

## 1.2 Thesis Contribution

The contribution of this thesis can be summarised as follows:

**1. A Comprehensive Multi-Stage Decision Workflow:**

By means of a sequential workflow, the system guides consumers through all stages of the deployment decision-making. Every stage gathers essential information, that influences the

final decision starting with defining urgency and agency-specific details, to specify technical and operational requirements. This approach, ensures, that all relevant factors are considered, and helps one to holistically assess challenging circumstances.

## **2. Integration of Argumentation-Based AI for Conflict Resolution in Deployment Choices:**

The system evaluates conflicting deployment requirements, using preference-based argumentation applied via Gorgias Cloud. By means of Raison AI, a set of Prolog rules, generated and maintained, formalises decision logic, so ensuring that technical constraints, cost efficiency, urgency, and scalability priorities are correctly balanced. When elements, like security against cost efficiency, or scalability against resource availability, oppose each other, the argumentative process lets conflict be resolved. This iterative thinking makes transparent, flexible, understandable decisions fit for hybrid cloud systems possible.

## **3. Automated Generation of Deployment Artifacts:**

Once the optimised deployment strategy is determined, the system automatically produces configuration files (e.g., YAML files), that are ready for immediate implementation. This automation, bridges the gap between high-level decision support and practical deployment, reducing manual errors, and streamlining the transition from planning to execution in DevOps (Development Operations) environments.

## **4. User-Friendly Web Interface for Non-Technical Decision-Makers:**

Knowing the different levels of knowledge among public sector players, the system features a user-friendly web-based interface, using Thymeleaf (Spring Boot), and RESTful APIs to communicate with the server. The interface consists of dynamic visuals, real-time input parameter validation, and methodical decision support, to enable users, understand the justification, behind every deployment recommendation. This design ensures, accessibility for policy-level decision-makers, as well as IT professionals, therefore encouraging group decision-making, in challenging cloud migration environments.

## **5. Real-World Validation in Cloud Environments:**

Designed especially for public sector installations, inside a Government Cloud (on-site data centres), the technology has been assessed using realistic case studies and pilot projects. These tests expose how well it manages particular difficulties, presented by hybrid infrastructures, including balancing public and on-site cloud systems. More transparency, better choices on deployment, and improved operational efficiency, all point to the real relevance of our method. This thesis offers a fresh, argumentation-based decision-support system that automates difficult cloud deployment decisions, in addition to fair, honest recommendations generally, combining advanced reasoning powers, with a multi-stage decision process.

# **1.3 Thesis Outline**

Each of the six major chapters that make up this thesis, is devoted to a distinct facet of the research and development process. The background information, context, and theoretical underpinnings required, to help the reader comprehend are provided in the first two chapters (Chapters 1 and 2). The system's design, implementation, and evaluation are covered in detail in the following chapters, and the final chapter offers a summary of the results and recommendations, for further research. Below is a more thorough description of the thesis structure.

## **Chapter 2 – Background**

Chapter 2 lays the groundwork for the research, by introducing the principles of argumentation theory and its applications in decision-support systems. This chapter reviews key literature on cloud deployment strategies, government cloud environments, and the challenges faced by public sector organisations. An overview of relevant frameworks, such as Gorgias Cloud, and the role of logic programming (with Prolog and Raison AI) in formalising decision criteria, is also provided. This background, sets the stage for understanding the advanced reasoning capabilities, integrated into our system.

## **Chapter 3 - Methodology and System Design**

In Chapter 3, the functional requirements and design specifications of the system, are presented in detail. This chapter describes the multi-stage decision workflow—from defining urgency details, infrastructure, location, resource, and scalability requirements, to the final combined decision and configuration file generation. The key features of each module are discussed, along with the methods used to capture and process the input parameters. This section also outlines how the system integrates with external reasoning engines, such as Gorgias Cloud, and the role of Prolog rules generated, by Raison AI in refining decisions.

## **Chapter 4 - Implementation**

Chapter 4, mostly addresses the design of the user interface and the whole user experience. We go into great length on the several points of view that of scalability, infrastructure choice, location specification, urgency, and resource needs. This chapter emphasises the design ideas, applied to guarantee the usability and accessibility of the interface, for both technical and non-technical users. It also covers the customising choices, which let users interact with the system, depending on their needs and degree of knowledge.

## **Chapter 5 - Evaluation**

It describes the evaluation methodologies used to test the system’s performance, usability, and effectiveness in real-world hybrid cloud environments. Detailed results from scenario-based tests and user evaluations are provided, along with an analysis of the strengths and limitations observed during the evaluation process.

## **Chapter 6 - Conclusions and Future Work**

The final chapter summarizes the key findings and contributions of the thesis. It reflects on the overall impact of the developed system, in automating complex deployment decisions, and discusses the benefits of using an argumentation-based approach. Additionally, this chapter outlines the limitations encountered during the research and proposes potential avenues for future enhancements, such as extending the decision model, incorporating new technological developments and exploring additional applications in related domains.

# Chapter 2

## Literature Review

This chapter reviews the corpus of recent research relevant to cloud computing, argumentation frameworks, and decision support systems. One can develop a theoretical foundation, for the project, by looking at previous research and developments in these areas. This study examines how argumentation theory has been incorporated into contemporary cloud installation decision-making processes, and how advancements in technology, have enhanced these processes. It also covers several models and tools, that have shaped the design, and application of automated decision systems historically. Examining present trends and approaches, helps this literature review to highlight gaps in current systems, that the present project aims to solve, so setting the scene for the creative elements of the thesis. This analysis, not only emphasises the importance of the project, in the framework of continuous research, but also helps to match it, with more general goals of enhancing efficiency, scalability, and user accessibility in cloud infrastructure decision making.

### 2.1 Argumentation

Argumentation logic is the multidisciplinary study of logical reasoning [3], as means of reaching conclusions. It covers both formal inference rules and pragmatic procedures, applied in daily decision making, and explores the approaches of debate, dialogue, and persuasion. Argument is essentially about building and assessing sets of ideas, that support or contradict a given point of view. Artificial intelligence [4] finds great use for this field, since it provides a basis for creating transparent and explainable decision-support systems, able of managing contradicting information. Applications for this rather new, fast-paced technology range from basic games to the medical domain [5], network security [6], cognitive assistants [7] [8] and business programs [3] [9].

Argumentation in decision-support systems, lets stakeholders formally express goals, constraints, and requirements as logical statements. These can be handled by automated reasoning engines to generate explainable, policy-compliant decisions, particularly in settings with tight operational or regulatory restrictions, such public sector cloud installations.

**Example.** To illustrate the concept of argumentation in practice, consider a scenario in the context of IT system deployment. Suppose a team must decide whether to migrate a public sector application to a cloud infrastructure. One argument ( $A$ ) in favor of migration is: “Moving to the cloud will improve scalability and reduce operational costs.” However, a counter-argument ( $B$ ) is: “Migrating to the cloud could compromise data privacy due to external data storage.” A supporting argument for  $A$  ( $AI$ ) might state: “The cloud provider

offers certified security measures.” On the other hand, a supporting argument for  $B$  ( $B1$ ) could assert: “Regulatory constraints require sensitive data to remain on-premise.”

This simple example demonstrates how arguments and counter-arguments can be explicitly stated and assessed, allowing for a transparent and structured approach to complex decision-making. In formal argumentation theory, such scenarios are often represented as directed graphs, where nodes denote arguments, and edges indicate attacks, or supports between them [10]. This graphical structure enables automated systems to systematically evaluate which arguments are ultimately accepted, rejected, or remain undecided.

Argumentation thus not only clarify the reasoning process for stakeholders, but also enables automated systems to generate, compare, and explain alternative courses of action. This transparency is especially valuable in domains where accountability and justification of decisions are paramount.

The main arguments and their relationships are as follows:

- **A1**: The cloud provider offers certified security measures (supports A)
- **A**: Moving to the cloud improves scalability and reduces cost
- **B**: Migrating to the cloud risks data privacy
- **B1**: Regulation requires sensitive data to remain on-premise (supports B)

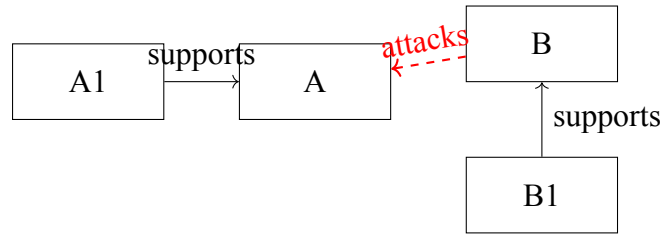


Figure 2.1: Simple argumentation graph for cloud migration decision. Solid arrows indicate support, dashed arrows indicate attacks.

In the graph, shown in Figure 2.1, argument A1 supports A, B1 supports B, and B attacks A. The structure allows the system to reason about which arguments are ultimately accepted, based on the strengths of the supports and attacks. An argument is considered “accepted”, if its supports outweigh the attacks against it, according to the system’s evaluation rules.

## 2.2 Preference-Based Argumentation

While general argumentation structures help to resolve logical conflict, preference-based argumentation ranks arguments, based on contextual preferences, so addressing more advanced forms of conflict. Preferences help a reasoning engine decide, which arguments are most relevant in a given context, when there are contradicting criteria, such as cost against security, or latency against scalability. In fields, like policy-making, and infrastructure choice, where stakeholder values and trade-offs exist, this approach performs particularly effectively. Thanks to preference-based models, which provide arguments ordering relations, systems can pick between conflicting conclusions, in a way, that is consistent with user-defined or policy-based priorities. A classic example is choosing between a public cloud (for economy) or a

service delivered on an on-site server (for data sovereignty). While both decisions have good reasons, preference rules, based on risk tolerance, cost sensitivity, or laws, like the GDPR, can help to resolve the conflict.

In formal systems, this is typically modeled by extending Dung’s framework with a set of preference rules:

If argument  $A$  is preferred over  $B$ , then  $A$  defeats  $B$  even if both are valid.

In computational models of reasoning, preference-based argumentation extends the classical abstract argumentation framework (AAF) by incorporating preferences among arguments. Originally introduced by Dung [10], an argumentation framework is defined as a pair:

$$AF = \langle A, R \rangle$$

where:

- $A$  is a set of abstract arguments.
- $R \subseteq A \times A$  is a binary relation representing attacks between arguments.

In a preference-based argumentation framework (PAF), the structure is extended to:

$$PAF = \langle A, R, \succ \rangle$$

where:

- $\succ$  is a preference relation over  $A$  (i.e.,  $a \succ b$  means  $a$  is preferred over  $b$ ).

This extension allows an argument  $a$  to defeat  $b$ , if  $a$  attacks  $b$  and is not less preferred than  $b$ . This mechanism is crucial, when modeling real-world decisions where not all arguments are equally valid.

## 2.3 Computational Tools for Argumentation

Argumentation is not just a theoretical idea, but also a useful toolkit accessible for daily application. Over the past decade, several frameworks have emerged to support computational argumentation, including platforms for preference-based reasoning, rule definition, and integration into web-based infrastructures. This section surveys the most relevant tools in this domain.

### 2.3.1 Prolog as a Foundation for Computational Argumentation

Originally designed in first-order logic, Prolog [11] is a declarative programming language, mainly used in symbolic thinking and artificial intelligence projects. Its syntax and inference rules, fit rather nicely for developing knowledge-based systems, and automated thinking engines. Prolog’s logic programming paradigm presents a simple approach to contain arguments, rules, facts, and inference procedures within the framework of computational argumentation.

One of Prolog’s main benefits is its built-in support of non-monotonic reasoning, since it meets the standards of argumentation systems. Several argumentation engines and frameworks, such as Gorgias [12] and DeLP [13], are implemented in Prolog or have adopted



Prolog-style rule-based logic, as their core. These systems leverage Prolog’s expressiveness, for defining argument schemes, attack/support relations, and preference handling, thereby enabling complex forms, of automated reasoning and justification. Prolog’s long-standing history in AI research, coupled with its flexibility in representing logical constructs, has established it as a foundational tool in the development and experimentation of computational argumentation.

**General Overview.** *Prolog programs are composed of facts, rules, and queries, supporting the representation of knowledge and automated reasoning through logical inference.*

#### **Basic Structure in Prolog Syntax:**

```
% Facts
fact1.
fact2.
fact3.

% Rules
option1 :- fact1, fact2, fact3.
option2 :- fact1, fact2.

% Preference Rule (meta-level, to choose option if both match)
prefer(option1, option2).

% Query
?- X.
```

This generic example shows how:

- Facts represent base information.
- Rules specify conditions for conclusions.
- Preferences indicate which result is favored, when multiple conclusions are possible.
- A query retrieves the preferred conclusion.

Prolog thus enables both logical inference and preference handling in decision-making scenarios.

### **2.3.2 Gorgias Cloud as a Tool for Argumentation-Based Reasoning**

Originally developed from the Prolog logic programming language [3], Gorgias Cloud is a web-based computational tool for argumentative based thinking. The system is designed to provide scenario modelling, execution, and explanation in contexts when explicit preferences and conflict resolution define decisions. Argumentation policies are encoded using the Prolog source files (*.pl*), which comprise *facts*, *rules*, and preference declarations. By use of modular arrangement, these files enable layered and reusable representations of policy logic, over several spheres of influence. The platform evaluates the active policy modules, during implementation by automatically resolving dependencies between rules and preferences, therefore generating transparent and well-founded judgements. Apart from web interface, Gorgias Cloud offers a *RESTful API* that facilitates the integration with automated processes

and outside applications, such those found in bigger decision support systems. By using Prolog possibilities of preference-based reasoning, Gorgias Cloud presents a strong framework for the formalisation, analysis, and justification of complex decision-making procedures.

### Modeling Argumentation Rules with Preferences in Gorgias

Rules and more complex preferences in Gorgias Cloud follow the structure:

$$rule(ID, Conclusion, Conditions)$$
$$rule(ID, prefer(A, B), Conditions)$$

where  $A$  and  $B$  are the identifiers of competing rules, and the preference is active, only if the given conditions are satisfied.

### Explanation and Transparency

Gorgias Cloud outputs include both internal logical trace and human-readable justifications. A typical, generic explanation may read:

*"Option1 is supported by the facts fact\_a  
and fact\_b,  
and is stronger than the argument for option2  
under the same conditions."*

This structured explanation promotes explainable AI (XAI), crucial in public sector decision-making, where accountability and traceability are required.

### Modularity and Transparent Scenario Analysis

A distinguishing strength of Gorgias Cloud, is its ability to deliver clear and transparent explanations, for every decision, tailored to the scenarios and policies encoded in each .p1 file. By allowing users to manage and switch between distinct policy modules, Gorgias Cloud not only supports scenario expansion and comparative analysis, but also ensures that each solution, is fully explainable and traceable to the underlying rules and preferences. This commitment to transparency makes Gorgias Cloud a valuable platform, for rigorous logic-based decision making, supporting both educational exploration and formal applications, where justification and accountability are paramount.

#### 2.3.3 Raison: No-Code Symbolic Argumentation-Reasoning Platform

Designed for users without programming knowledge, *rAison* is a no-code tool, for symbolic AI rule development. It allows domain experts, to create structured templates, from which they can encode policy and decision logic, subsequently compiled into Prolog-compatible rules. For public managers and policy makers, who might not know logic programming, this greatly reduces the entrance barrier.

Users can create their own scenarios by modelling, in natural language, the facts and logical outcomes, ultimately producing their own .p1 files. Each scenario consists of one or more conditions, called *elements*, combined with possible conclusions, or *options*. These

visual configurations, shown in Figure 2.2, represent logical rules, that define policies in a structured, explainable way. This kind of framework makes it easy to generate and manage large sets of rules, even across complex policy spaces.

Furthermore, it is offering a strong interface for conflict resolution. Figure 2.3 show how the system automatically detects overlapping or contradicting rules, and lets the user resolve them, by just choosing the preferred scenario with a click. Once resolved, the system guarantees consistency across the decision model, and allows users to iterate, refine, and export their reasoning as Prolog code.

Finally, as illustrated in Figure 2.4, *rAison*, enables users to run their configured decision models in practice. For instance, when applied to the problem of defining timeline urgency, users input a set of factual data (such as contract status or request type) and execute the model. The system then, evaluates all relevant rules and visually presents the resulting urgency level(s), through an interactive, user-friendly interface. This seamless integration of rule definition, conflict resolution, and live querying makes *rAison* a powerful tool for rapid development and deployment of explainable decision support systems.

	urgency(normal)	urgency(high)	urgency(urgent)
request_type(mog) contract(no) urgentBasedOnDate	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
agency_category(independentAuthority) contract(no) urgentBasedOnDate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
agency_category(localGovernment) contract(no) urgentBasedOnDate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
request_type(other) highBasedOnDate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
urgentBasedOnDate	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
contract(no)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
request_type(mog)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
agency_category(localGovernment)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 2.2: Visual rule definition in *rAison*: Scenarios mapped to options.

Select a scenario to resolve a conflict

8 / 8 CONFLICTS SOLVED

solved ✓  
agency\_category(independentAuthority) contract(no) urgentBasedOnDate

solved ✓  
contract(no) agency\_category(localGovernment) urgentBasedOnDate

solved ✓  
contract(no) agency\_category(localGovernment) urgentBasedOnDate request\_type(other)

solved ✓  
agency\_category(independentAuthority) contract(no) urgentBasedOnDate request\_type(other)

Figure 2.3: Conflict resolution interface: Choosing the preferred scenario to ensure consistent reasoning.

The screenshot shows a web application interface with the following components:

- Header:** A hamburger menu icon, the text "urgency24022025", "Run application", a "NATURAL LANGUAGE" toggle switch, and a help icon.
- Choose input data:** A section with a "Data" label and a dropdown menu containing three selected items: "highBasedOnDate", "contract(no)", and "request\_type(other)".
- Results:** A section containing three dropdown menus, each with a status icon and a conclusion:
  - First dropdown: A green checkmark icon and the conclusion "urgency(normal)".
  - Second dropdown: A green checkmark icon and the conclusion "urgency(high)".
  - Third dropdown: A red "X" icon and the conclusion "urgency(urgent)".
- Buttons:** A yellow "RUN" button is located to the right of the input data dropdown.

Figure 2.4: Querying a rule-based decision model: Given a set of input facts, the system evaluates the corresponding rules and returns the derived conclusion.

## 2.4 Technologies for Web Application and Integration

Modern web applications are built using a wide range of technologies designed to make them more scalable, compatible, and connected. Recent advancements in frameworks, protocols, and standards have made it much easier for web apps to work smoothly with other online services. By understanding the key concepts and keeping up with the latest trends in these technologies, developers can create solutions, that are flexible and strong enough to meet the ever-changing demands of today’s software environments.

Using a common set of guidelines derived from the HTTP protocol, representational state transfer APIs, also known as RESTful APIs (Application Programmable Interface), enable online applications and systems to communication. Through HTTP requests, these APIs facilitate, the easy access and management of resources, such as data models or computing services. REST is widely used in modern software and web development, due to its ease of use, scalability, and compatibility with current web standards. To develop or consume RESTful APIs, developers often rely on supporting frameworks and tools.

One widely used framework in the Java ecosystem is **Spring Boot**, which simplifies the creation of RESTful web services. It provides built-in support for REST controllers, dependency injection, and data serialization, allowing developers to easily define endpoints, handle requests, and return structured responses.

**Swagger** (now part of the OpenAPI initiative) is commonly used to document and test RESTful APIs. It generates machine-readable specifications, that describe API endpoints, input parameters, response formats, and authentication methods. Tools, like Swagger Editor and Swagger UI, allow both human-readable and interactive API exploration, supporting rapid development and integration.

### 2.4.1 Technical Background: HTTP Methods and JSON Format

RESTful APIs communicate using a standard set of HTTP methods, each of which corresponds to a specific type of operation on a resource, as shown in Table 2.1.

These methods are used in conjunction with structured request bodies, most often formatted in **JSON** (JavaScript Object Notation). JSON is a lightweight, text-based format for representing structured data as key-value pairs. It is widely used due to its readability and compatibility with many programming languages.

Table 2.1: Common HTTP Methods in RESTful APIs

Method	Description	Typical Use Case
GET	Retrieve resource	Fetch user data
POST	Create resource	Submit a new form
PUT	Update entire resource	Update user profile
PATCH	Update part of a resource	Change user email
DELETE	Remove resource	Delete user account
OPTIONS	Get available methods	Check API capabilities

An example JSON payload used in an HTTP request might look like:

```
{
  "user": "agent007",
  "action": "submit_request",
  "parameters": {
    "priority": "high",
    "document": "report2025.pdf"
  }
}
```

This object includes multiple fields to be interpreted by the server: an identifier, a requested action, and relevant parameters. Such a payload, would typically be sent in a POST request, to a specified API endpoint. When designing RESTful APIs, it is important to follow best practices, such as consistent naming conventions for endpoints, appropriate use of HTTP status codes, shown in Table 2.2, stateless communication, and clear error messaging. These principles, enhance interoperability and ease of use, for both developers and clients consuming the API.

Table 2.2: Common HTTP status codes for RESTful API responses

Code	Name	Description
200	OK	The request was successful and a response is returned.
201	Created	A new resource has been successfully created.
400	Bad Request	The request could not be understood or was missing required fields.
401	Unauthorized	Authentication is required or has failed.
404	Not Found	The requested resource could not be found.
500	Internal Server Error	The server encountered an unexpected condition.

Figure 2.5 illustrates a typical communication flow between a client and a REST API server. The client sends an HTTP request (such as GET or POST), and the server responds, often with JSON-formatted data.

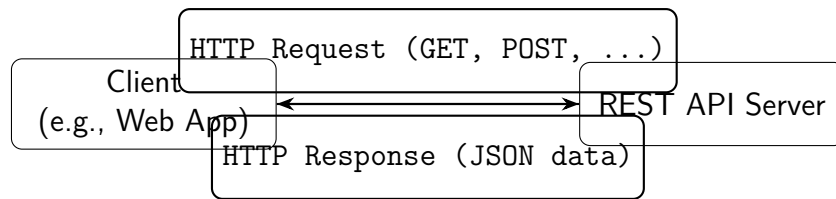


Figure 2.5: Basic REST API Communication Flow

## 2.4.2 Spring Boot Framework for Web Service Development

Spring Boot is a widely adopted framework, built on top of the Spring ecosystem for developing production-ready, stand-alone Java applications. It significantly simplifies the process of building web services and APIs by providing default configurations and embedded servers, such as Tomcat. Given its lightweight design, extensive community support, and compatibility with REST and Swagger, Spring Boot is an excellent choice for creating workflow-driven processes and web applications. It can also be used to develop REST clients that communicate with external APIs (such as the Gorgias Cloud API), encapsulating reasoning logic within a modern, maintainable software architecture.

### Core Architecture and Features

At its foundation, **Spring Boot** builds upon the established principles of the Model-View-Controller (MVC) architectural pattern, effectively separating concerns among request handling, business logic, and data management.

One of the framework's primary advantages is its **auto-configuration** capability, which dynamically configures application components based on detected dependencies. This reduces boilerplate code and significantly accelerates the development process. In addition, Spring Boot streamlines deployment by providing an **embedded server**, most commonly Tomcat or Jetty, allowing applications to execute independently, without reliance on external servlet containers.

Development within Spring Boot is characterized by an annotation-driven approach. The use of declarative constructs, such as `@Controller`, `@Service`, and `@Repository`, enables clear delineation of roles and wiring within the application. This annotation-centric paradigm not only promotes clarity, but also enhances modularity, reinforcing the principle of **separation of concerns** by organizing logic into discrete layers.

Such modular design improves both maintainability and scalability, especially in complex systems. Furthermore, Spring Boot offers robust **integration support**, enabling seamless connectivity with external systems and APIs, and thereby facilitating the development of sophisticated business workflows.

Figure 2.6 illustrates the typical architecture of a Spring Boot web application. HTTP requests from the user or client are processed by the Controller layer, which delegates business logic to the Service layer. The Service layer may access persistent data through the Repository layer, or call external APIs, such as the Gorgias Cloud API. For web applications with dynamic content, the Controller can also pass data to the View layer (e.g., Thymeleaf templates) for rendering HTML responses. This separation of concerns promotes maintainability, scalability, and clear code organization.

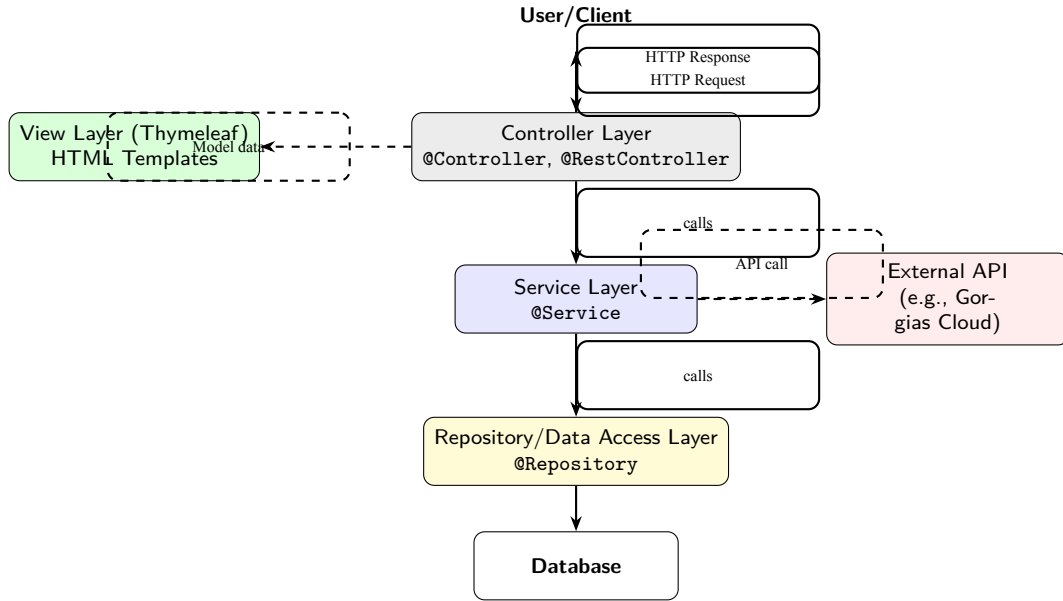


Figure 2.6: Layered architecture of a Spring Boot web application, showing main components, user flow, database, and integration with external APIs, such as Gorgias Cloud.

## Comparative Analysis

Spring Boot is recognized in the literature as a leading solution for building scalable, maintainable web services and microservice architectures in Java [14], [15]). Compared to traditional Java EE development, Spring Boot’s auto-configuration and annotation-driven model have been shown to significantly reduce boilerplate and accelerate time-to-deployment [16].

A number of studies and technical reports (e.g., [17, 18]) emphasize Spring Boot’s suitability for microservices, particularly when paired with Docker or cloud orchestration tools, such as Kubernetes. Its strong integration with RESTful web services, combined with mature support for API documentation (e.g., Swagger/OpenAPI), make it a frequent choice in both enterprise and public sector contexts.

Several comparative analyses [19] highlight the advantages of Spring Boot over other frameworks, such as Node.js/Express, Django, or ASP.NET Core, especially in environments where reliability, security, and JVM compatibility are paramount.

Best practices documented in the literature include strict separation of concerns across MVC layers, comprehensive unit and integration testing, and use of CI/CD pipelines for deployment [20]. However, some challenges are noted, such as managing complex configurations in large-scale systems and ensuring optimised resource utilization.

## 2.5 Dynamic Web Content Thymeleaf Template Engine

Designed for both web and standalone environments, **Thymeleaf** is a contemporary server-side Java template engine [21]. It easily interacts with the Spring ecosystem to allow server-side template processing to produce HTML content dynamically. Though not produced within a running application, Thymeleaf’s “natural templating” approach lets its templates remain valid HTML, viewable and editable in browsers or IDEs. This tool simplifies front-end designers’ and backend developers’ working together [22].

Thymeleaf often acts as the View layer in Spring Boot applications, compiling View layer data from controllers, by isolating presentation logic from business and data logic [23], into structured HTML for client responses. This architectural separation maintains the ideas of the Model-View-Controller (MVC) paradigm. Thymeleaf gives flexibility for developing complex, data-driven interfaces, so supporting conditional rendering, iteration, and internationalisation. Its compatibility with HTML5 and extensible dialect system adds even more value for current web development [22].

As seen in Figure 2.6, the combined use of Thymeleaf and Spring Boot promotes rapid prototyping, maintainability, and efficient form handling, so supporting strong and scalable web applications.

## **2.6 Deployment Technologies: Docker and Git**

### **Docker: Containerization for Consistent Deployment**

Docker is an open-source platform that provides a robust containerization solution, ensuring consistent deployment across different environments [24]. By encapsulating the application and its dependencies within containers, Docker eliminates the “it works on my machine” problem that often plagues software deployment processes [25].

#### **Capabilities and Benefits**

Docker containers package all dependencies, libraries, and configuration files needed for the application to run, ensuring identical behavior across development, testing, and production environments.

Unlike traditional virtual machines (VMs), Docker containers share the host OS kernel, resulting in lightweight deployment units, that consume fewer resources, while maintaining isolation, between applications [26]. This lightweight architecture facilitates horizontal scaling and enables seamless deployment, across cloud providers and on-premises infrastructure. Through Docker Compose and container orchestration tools, complex multi-container deployments can be automated, significantly reducing manual configuration and potential human errors during the deployment process.

#### **Limitations and Challenges**

Although Docker offers significant benefits, there are notable limitations. Security concerns have been raised, due to the shared kernel architecture, and some performance overhead may be present in certain scenarios [27]. Additionally, the learning curve associated with container orchestration and configuration can be a barrier for new users.

Compared to other tools, such as Podman or orchestration platforms like Kubernetes, Docker stands out for its simplicity and widespread adoption, but larger-scale systems may require the advanced orchestration features provided by Kubernetes [28].

### **Docker Command Line Implementation**

Docker provides a Command Line Interface (CLI) that enables users to build images, run containers, and manage application environments directly from the terminal. The following



commands, as shown in Code 2.1, demonstrate common Docker CLI usage for essential container management tasks:

```
# Build an image from a Dockerfile
docker build -t my-image:latest .

# List all local images
docker images

# Run a container from an image
docker run -d -p 80:80 my-image:latest

# List all running containers
docker ps

# Stop a running container
docker stop <container_id>

# Remove a container
docker rm <container_id>
```

Code 2.1: Key Docker Commands for Container Management

### Explanation of the Commands:

- `docker build -t my-image:latest .`  
Builds a Docker image from the Dockerfile in the current directory.  
The `-t` flag assigns the image the tag `my-image:latest`.
- `docker images`  
Lists all Docker images stored locally on the system.
- `docker run -d -p 80:80 my-image:latest`  
Starts a new container from the `my-image:latest` image.  
The `-d` flag runs the container in detached (background) mode.  
The `-p 80:80` option maps port 80 of the host to port 80 of the container.
- `docker ps`  
Displays all currently running containers.
- `docker stop <container_id>`  
Stops a running container specified by its container ID.
- `docker rm <container_id>`  
Removes a stopped container from the system.

In summary, Docker's CLI, combined with its comprehensive ecosystem, provides precise control over container configurations and enables automated deployment procedures, making it a popular choice for consistent and reproducible software deployment.

## Git: Version Control and Collaborative Development

Git is a widely used distributed version control system, designed to manage source code changes efficiently, supporting collaborative development and ensuring code integrity [29] [30]. Its distributed architecture, allows multiple contributors to work on different features simultaneously, enhancing project scalability and reliability [31].

## **Capabilities and Benefits**

Git's distributed design, enables team members to work concurrently on separate branches, without interference, significantly accelerating the development cycle, while maintaining high code quality. Through branch-based workflows, development efforts can be isolated and systematically merged after review, ensuring that only tested and approved code reaches production. Additionally, Git's comprehensive history tracking provides a complete audit trail, facilitating quick identification of issues and rapid rollback to stable versions when necessary.

## **Limitations and Considerations**

Despite its strengths, Git has a learning curve for new users, especially regarding branch management and conflict resolution [31]. Alternative version control systems, such as Subversion and Mercurial, are available, but Git is preferred in most modern projects due to its flexibility and robust ecosystem [30].

## **Git Command Line Workflow**

The Git Command Line Interface (CLI) equips developers with versatile tools to manage code versions, collaborate with teams, and streamline the software development lifecycle. Below, in Code 2.2 are some essential Git commands for effective version control are shown:

```
# Create a new feature branch for development
git checkout -b feature/new-feature

# Commit changes with a descriptive message
git commit -m "Describe the implemented changes"

# Push local changes to a remote repository
git push origin feature/new-feature

# Switch to the main branch
git checkout main

# Merge feature branch changes into the main branch
git merge feature/new-feature
```

Code 2.2: Common Git Commands for Workflow

### Explanation of the Commands:

- `git checkout -b feature/new-feature`  
Creates and switches to a new branch named `feature/new-feature`, supporting isolated feature development.
- `git commit -m "Describe the implemented changes"`  
Records changes to the repository with a descriptive commit message, aiding in traceability.
- `git push origin feature/new-feature`  
Uploads the local branch to the remote repository, facilitating collaboration.
- `git checkout main`  
Switches to the main branch for merging or deployment.
- `git merge feature/new-feature`  
Integrates changes from the feature branch into the main branch, combining new work with the production codebase.

In summary, Git's distributed model, powerful branching, and reliable history tracking have made it the industry standard for modern version control and collaborative software development.

## Chapter 3

# Methodology and System Design

Successful Public Sector deployment of cloud-based systems, depends on both methodological and technological innovation. This chapter presents the method and system architecture developed, to automate decision-making on deployment for the Ministry of Digital Governance. Based on a thorough requirement analysis, the approach transforms complex operational parameters, including **security** limitations, **performance** criteria, and **resource** allocation, into a disciplined process of decision-making. Developed in close coordination with ministry stakeholders, these requirements go beyond mere guidelines, to provide the structural foundation of all system design procedures followed.

Whether it relates to memory use, regulatory compliance, or predicted computing load, every parameter is investigated for its impact on system behaviour, scalability, and compliance with government rules. Our method sees needs as dynamic, interactive variables within a living deployment ecology, rather than as fixed checklists. This lens guarantees, that every deployment considers the priorities and constraints of the real world by including performance, resource, and security needs into a complete decision model.

A fundamental part of the approach, is turning these challenging specifications into manipulable "system facts" via programming. By consuming and analysing unprocessed user data via a structured interface, a special middleware layer, included into Java Spring Boot, enables this modification. The user interface of the system, shown in Figure 3.4, acts as a link between technical backend and human decision-makers. Whether a user's choice indicates budgetary priorities, data sensitivity, or scalability needs, it is immediately encoded as a logical, objective data point. This process, blends the naturally subjective aspect of policy-driven decision-making, with the objectivity required for automated reasoning.

Prolog facts upon form submission, dynamically parse and reformulate user inputs, hence creating a seamless path from interface to inference engine. Communication with the Gorgias Cloud platform, is based on these logical realities, where advanced frameworks for reasoning evaluate and balance opposing needs. Thanks to its connection with Gorgias Cloud, the system can travel complex, often contradicting decision domains and produce technically sound, clearly understandable recommendations.

The system also leverages Raison AI to refine rule sets and enhance decision-paths. By using no-code symbolic thinking, Raison AI guarantees, that the decision engine will remain relevant, as needs evolve and so enhances the comprehensiveness and adaptability of the logic model. This multi-layered architecture, which combines middleware data processing, user-centric interfaces, logic programming and middleware data processing, represents a rigorous and quite practical approach.

Ultimately, this methodological approach guarantees that exact, practical deployment

plans fairly represent high-level user needs. Combining domain knowledge, formal logic, and automated reasoning generates a solution that facilitates scalable, efficient, and transparent cloud adoption, inside the specific framework of the Greek public sector.

### 3.1 System Architecture Overview

This section provides, a detailed architectural overview of the decision support system, designed for cloud deployment. The system architecture is designed to convert user inputs into actionable decisions using a combination of YAML file generation, Prolog fact conversion, and query execution, through the Gorgias Cloud Platform. The process flow depicted in the block diagram of Figure 3.1 illustrates the sequential steps taken from receiving user input to delivering final decision outputs.

#### Process Description:

1. **User's Input:** Initially, users provide input through a web interface. This input typically consists of specific parameters related to cloud deployment needs, such as scalability, budget, data sensitivity, and infrastructure control.
2. **Converting Input to Prolog Facts:** The input data, is then processed to transform it into a format, that can be understood by the decision-making engine, Prolog facts. This transformation is crucial, as it ensures that the inputs are correctly interpreted by the underlying logic of the decision system.
3. **Query Execution:** Once the data is converted into Prolog facts, queries are formulated and executed on the Gorgias Cloud Platform. This platform uses a sophisticated argumentation, to evaluate the facts against predefined rules and scenarios.
4. **Results:** The outcome of the queries, which are the reasoned decisions, are then fetched from the Gorgias Cloud Platform. These results, determine the most suitable cloud deployment strategies, based on the initial user inputs and the logical deductions performed by the system.
5. **Converting Non-Human Explanation to Readable Decision:** In the final step, the system converts the technical outputs from the Gorgias Cloud into human-readable decisions. This conversion is essential for end-users to understand the reasoning behind each suggested deployment strategy.
6. **Optional YAML File Generation:** Once results are available and interpreted, there is an option for users to generate a YAML file. This file provides a structured representation of the decisions, facilitating further analysis or integration with other systems.

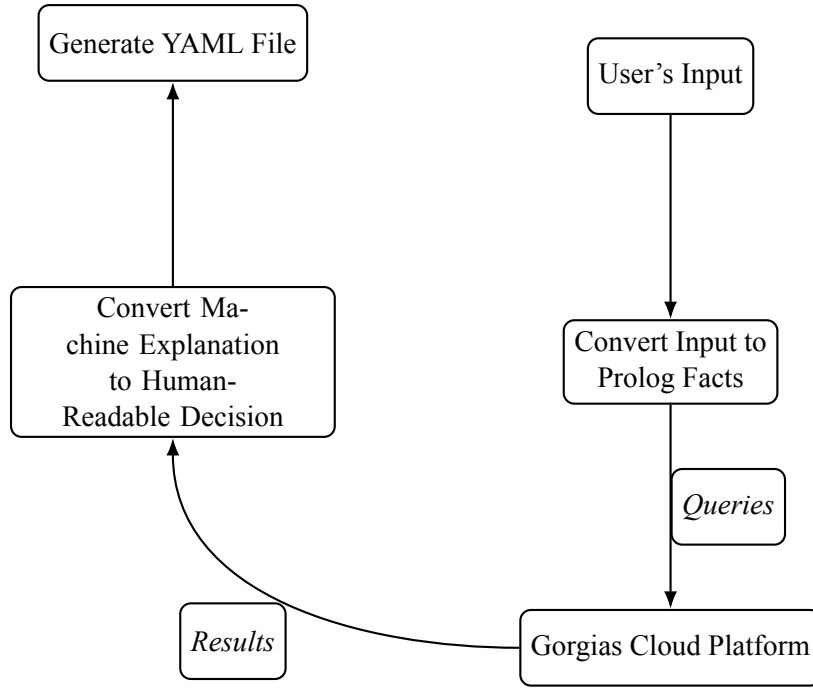


Figure 3.1: High-level system flow: User input is transformed into Prolog facts, processed via the Gorgias Cloud Platform, and the output is interpreted into human-readable decisions and optionally exported as a YAML file.

## 3.2 System Workflow and Behavior

To clarify the workflow depicted in Figure 3.1, we provide a complete example that demonstrates, the transformation and representation at each stage, from the initial user input, to the final YAML file.

**1. User Input** The process begins with a user-friendly web form, designed to capture essential parameters from public sector stakeholders. These inputs, guide the system, in determining the urgency level of the deployment timeline. An example of this form is shown in Figure 3.2.

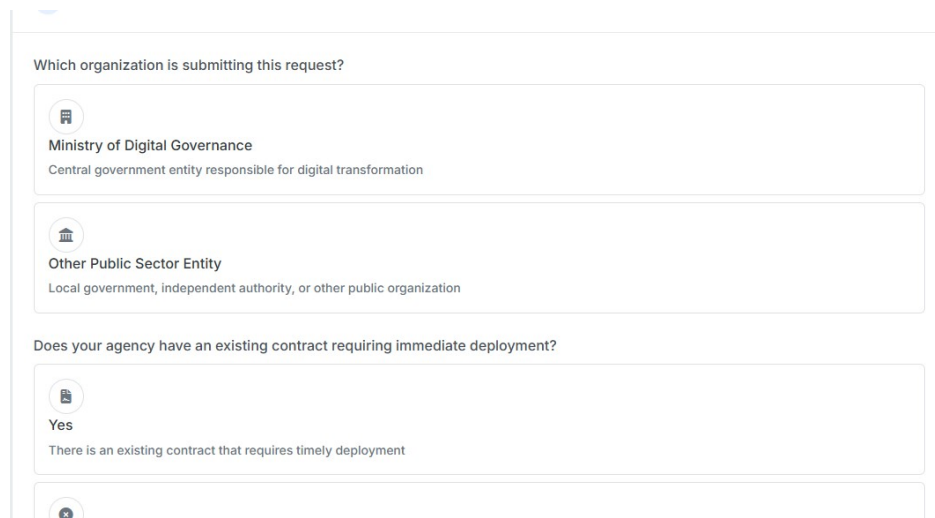
- **Agency Category:** Other
- **Contract with Contractor:** No
- **Planned Start Date:** mm-dd-yyyy

For demonstration purposes, we focus on the first web form, rather than all five forms that correspond to the system's core parameter categories. By combining three key fields, the system automatically infers an urgency level and prioritizes requests accordingly. Each parameter contributes its own assessment of urgency and the system resolves potential conflicts.

These parameters were defined in consultation with the supervising authority, the Ministry of Digital Governance, and serve as critical inputs to the urgency classification process. Notably, if the request originates from the Ministry of Digital Governance itself, it is automatically assigned the highest urgency level due to its institutional priority. In contrast, requests from other agencies are evaluated based on additional factors.

Specifically, the combination of agency type, contractual obligations, and the proximity of the planned start date, determines how time-sensitive the request is. This logic is grounded in formal policy rules set by the Ministry.

- If the request concerns an Integrated Information System of the Ministry and is of immediate priority, it is classified as **Urgent** and is executed within 3 days.
- If the request must be implemented immediately due to existing contracts and strict schedules, it is marked as **High** and is fulfilled within 10 days.
- If the request does not involve a strict timeline, it is labeled as **Normal** and is processed within 2 months.



The image shows a user interface for selecting urgency parameters. It consists of three main sections, each with a question and two radio button options. The first section asks 'Which organization is submitting this request?' with options for 'Ministry of Digital Governance' (Central government entity responsible for digital transformation) and 'Other Public Sector Entity' (Local government, independent authority, or other public organization). The second section asks 'Does your agency have an existing contract requiring immediate deployment?' with options for 'Yes' (There is an existing contract that requires timely deployment) and 'No' (There is no existing contract that requires timely deployment). The third section is partially visible and asks 'Does your agency have a strict timeline?' with options for 'Yes' and 'No'.

Figure 3.2: User Interface for Urgency Parameter Selection — aligned with Ministry rules for urgency classification

Some input fields, such as the **Planned Start Date**, require additional pre-processing, before they can be translated into logical facts. For this reason, we implemented a supporting Java routine, shown in Code 3.1, inside our Back-end framework, that computes the difference between today's date and the planned start date, and classifies the result into urgency levels, based on the date. The output is used to generate a corresponding Prolog fact:



```

import java.util.Date;
import java.util.concurrent.TimeUnit;

/**
 * Computes an urgency category from the temporal distance (in
 * days)
 * between a planned start date and the current date.
 *
 * @param plannedStart the date entered by the user
 * @param now          the moment at which the calculation is
 *                    performed
 * @return             one of "URGENT", "HIGH", or "NORMAL"
 */
private static String calculateUrgency(final Date plannedStart,
    final Date now) {
    final long millisBetween = plannedStart.getTime() - now.
        getTime();
    final long daysBetween = TimeUnit.DAYS.convert(
        millisBetween, TimeUnit.MILLIseconds);

    if (daysBetween <= 3) {
        return "urgentBasedOnDate";
    } else if (daysBetween <= 10) {
        return "highBasedOnDate";
    }
    return "normalBasedOnDate";
}

```

Code 3.1: Computation of an urgency class based on the time interval between the planned start date and the evaluation moment.

- Dates within the next 3 days are marked as `urgentBasedOnDate`.
- Dates between 3 and 10 days ahead are marked as `highBasedOnDate`.
- Dates more than 10 days away are marked as `normalBasedOnDate`.

**2. Conversion to Prolog Facts** The system uses a Java-based preprocessing layer to transform user-submitted form data into structured Prolog facts, which are then passed to the reasoning engine for evaluation. A sample of Java code, that performs this conversion can be found in Code 3.2.

```

try {
    // Add agency category fact
    String request = form.getRequest();
    if (request != null && !request.isEmpty()) {
        facts.add("request(" + request.toLowerCase() + ")");
    }

    // Add contract with contractor fact
    String contract = form.getContract();
    if (contract_with_contractor != null && !
        contract_with_contractor.isEmpty()) {
        facts.add("contract_with_contractor(" +
            contract_with_contractor.toLowerCase() + ")");
    }

    // Add highBasedOnDate fact
    String highBasedOnDate = form.getHighBasedOnDate();
    if (highBasedOnDate != null && !highBasedOnDate.isEmpty()) {
        facts.add("highBasedOnDate(" + highBasedOnDate.toLowerCase
            () + ")");
    }
}

```

Code 3.2: Java code snippet for transforming form data into Prolog-style facts

## Fact Generation Output

This transformation, derives valid Prolog facts, directly from the user input. Gorgias Cloud receives these facts, for further evaluation:

```

request(other).
contract_with_contractor(no).
highBasedOnDate.

```

Once the facts are generated, they are passed to the reasoning engine, which operates on the Gorgias Cloud infrastructure. Specifically, the system consults the appropriate ‘.pl’ (Prolog) file that encapsulates the policy rules defined for this case. These files have been created through Raison AI platform.

In this context, the engine infers the appropriate urgency classification, based on the input parameters. The result is returned in a structured format and can take one of the following values: **Urgent**, **High**, or **Normal**, depending on factors, such as planned start date, agency category, and contractual constraints.

**3. Query to Gorgias Cloud Platform** Along with these facts, the system generates a query for each case and sends it to the Gorgias Cloud Platform for reasoning. A typical query, for this case, might look like the one in Code 3.3:

```

gorgiasQuery.setQuery("urgency(X)");

```

Code 3.3: Setting a logic query string in the Gorgias request object.

**4. Reasoned Decision (Results)** The Gorgias Cloud Platform applies the policy rules to the input facts and query to infer a result. While one might expect the reasoning engine to simply return a direct value, such as

X = high

the actual output from the Gorgias Cloud Platform is considerably more detailed and comprehensive. Instead of a plain label, the system generates a structured response object that encapsulates not only the computed result, but also a detailed explanation of the underlying reasoning process. As shown in Code 3.4, the returned object includes multiple fields: an explicit indication of success, an array of rule identifiers contributing to the result, and, crucially, a *human-readable explanation* that details the logical chain leading to the final decision. This explanation clarifies, for example, why a particular urgency level was selected, by referencing both supporting and competing arguments, along with the specific facts that tipped the balance. The system thus provides full transparency into its reasoning, enabling traceability and accountability in deployment recommendations, a feature essential for Public Sector automation and compliance.

**5. Converting “Non-Human” Explanation to Readable Decision** To make it more accessible and understandable, the system transforms the ordered, “non-human” or better technical output, from the Gorgias Cloud reasoning engine into an explanation understandable to users. The raw query result contains internal rule references, variable bindings, and tough to directly grasp logic chains. Gorgias Cloud provides a so-called *human explanation*, however non-technical decision-makers could not be able to access this output right away, since it is still essentially technical in character. Extra layer of processing is done, to reformulate these technical explanations, into messages accessible to policy makers and end users, unfamiliar with logic programming. This process, is crucial to ensure that the recommendations of the system are not only visible and clear, but also really relevant to Public Sector environments, where users’ technical competency could vary significantly.

For instance, the result in Code 3.4 is returned from the reasoning engine:

```

Query Result: class GorgiasQueryResult {
  hasError: false
  hasResult: true
  result: [class QueryResult {
    explanation: [c3, p7, r8]
    explanationRulesHeadWithoutVariables: [p7, c3, r8]
    explanationStr: [c3,p7,r8]
    humanExplanation: 'Application Level Explanation

    The statement "urgency(high)" is supported by:

      - "request_type(other)"

    This reason is:
      - Stronger than the reason of:
        "request_type(other)" supporting "urgency(normal)"
        when:
          "request_type(other)"
          and
          "highBasedOnDate"

    Variables: {X = high}
  }]
}

```

Code 3.4: Query result from the reasoning engine

This structured output is automatically parsed and reformulated into an explanatory message intended for end users and decision-makers. For the example above, the translated explanation becomes:

*High Priority: Accelerated deployment required within 10 days. Prompt resource allocation recommended. Standard procedures should be followed with expedited processing.*

In addition, the system provides a rationale:

*Why this choice: High priority is stronger than Normal priority, and is supported by the fact that the request is from another ministry or organization, combined with the short time frame until deployment.*

This approach enables traceable, explainable AI (XAI) by bridging formal logic and policy-based inference with intuitive summaries, enhancing both trust and accountability in automated decision-making processes. An example of the corresponding user interface output is shown in Figure 3.3

**6. Finalizing Decisions and Applying Selections** Once all five decision cases, illustrated in Figure 3.4, have been processed, the system recommends appropriate options, based on the parameters provided for each case. At this stage, the user is empowered to make a final selection from the computed results, in order to finalize the deployment strategy and generate the corresponding configuration file.

## 1 SOLUTION

**High Priority: Accelerated deployment required within 10 days. Prompt resource allocation recommended. Standard procedures should be followed with expedited processing.**

↑ HIGH

### ❗ WHY THIS CHOICE?

- The request is from another ministry or organization.
- The deployment date is approaching soon, indicating a high priority.
- High priority is stronger than Normal priority, also supported by the fact that the request is from another ministry or organization, because the deployment date is approaching soon.

Figure 3.3: User Interface output showing the recommended urgency classification

After reviewing the available choices, as shown in Figure 3.5, the user clicks *Apply Selections* to confirm the chosen deployment strategy. This action prepares the system to generate the corresponding YAML configuration file.

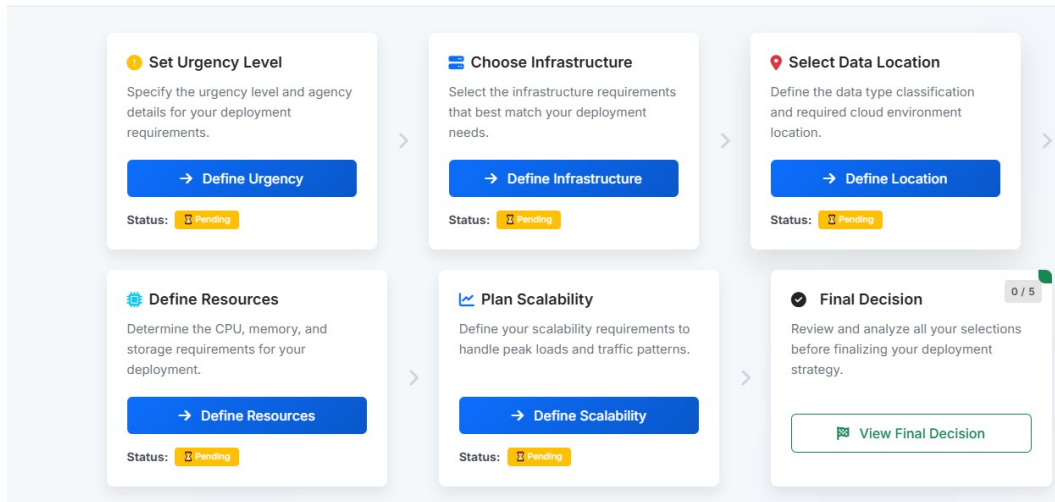


Figure 3.4: Overview of the user workflow for case-based decision support.


Figure 3.5: Interface showing applying final selections


**7. Recommended Strategy and YAML File Generation** This recommendation includes the selected deployment strategy, technical reasoning for the choice, and a YAML configuration file, tailored to the user's specific requirements.

In the example illustrated in Figure 3.6, the system recommends **Azure Functions (Premium)**. This strategy, offers enhanced serverless computing, with dedicated resources, VNet integration, and pre-warmed instances, ideal for performance, and critical event processing.


The recommendation is justified by technical reasoning, that takes into account the user's performance, latency, and disaster recovery requirements. As shown in Figure 3.7, the system identifies serverless computing, low latency needs, and high disaster recovery expectations, as the key decision factors.

Finally, the user is given the option to export the recommendation as a YAML configuration file. This file captures the decision outcome in a structured format and can be used for deployment automation or documentation purposes. An example YAML file for the recommended Azure Functions deployment is shown in Code 3.5.:


 Recommended Deployment Strategy





Azure Functions (Premium): Enhanced serverless compute with dedicated resources, VNet integration, and pre-warmed instances for performance-critical event processing


 This recommendation is based on your specific requirements for availability, performance, budget, and technical constraints.


★ Key Features of This Deployment Strategy

 Multi-region deployment

 Automated failover capability

 High availability architecture

 Comprehensive backup strategy

 Optimized performance




 Business continuity support


Figure 3.6: Recommended Deployment Strategy: Azure Functions (Premium)


 Decision Reasoning

Why This Strategy Was Chosen

Our analysis of your requirements led to this recommendation based on the following key factors:

 Technical Reasoning

 Serverless Computing: Allows code execution without managing underlying infrastructure.

 Low Latency Required: Applications requiring minimal processing and response delays.


 High Disaster Recovery: Comprehensive backup, replication, and failover capabilities for critical workloads.

Figure 3.7: Technical Reasoning Behind the Deployment Decision

```

# Use case: Event-driven applications requiring consistent
# performance, VNet integration, and longer execution times

apiVersion: 2023-01-01
kind: AzureDeployment
metadata:
  name: premium-functions
  description: Azure Functions Premium deployment
  for high-performance event processing
  tags:
    environment: production
    workloadType: serverless-premium
    costCenter: IT-Applications

parameters:
  location:
    type: string
    defaultValue: westeurope
    allowedValues:
      - westeurope
      - northeurope
      - eastus
      - westus2
    description: Azure region for resource deployment

  functionAppName:
    type: string
    defaultValue: func-premium
    description: Name of the function app (must be globally
      unique)

```

Code 3.5: Example Configuration File Structure (YAML)



This example demonstrates how the system moves from logical reasoning and policy evaluation to a concrete, executable configuration, completing the end-to-end decision-making workflow.

### 3.3 User Interface Design

The structural and functional design of the system components, produced for this system, is covered in this part. The modular implementation of data processing, user interaction, and decision logic, underlines their support of the choice of deployment strategy. The mechanisms by which users input guided forms, how these inputs are handled, and how decisions are derived depending on predefined criteria are covered in the next subsections.

#### 3.3.1 User-Guided Configuration through Decision Forms

This section presents the User Interface (UI) and the process management logic. The system is structured around five distinct forms, each corresponding to a critical decision-making category: urgency, infrastructure, location, resources, and scalability. Each form is designed to collect targeted information, relevant to its respective domain, thereby enabling structured, data-driven decisions. By isolating these categories, the UI ensures a transparent, modular, and traceable workflow, allowing users to engage with each aspect of the deployment strategy, independently and systematically. Furthermore, the UI is implemented using Thymeleaf and HTML, enabling dynamic form rendering, real-time validation, and seamless interaction with backend services.

- **Urgency:** The urgency input, allows the system to determine, the appropriate deployment timeline. Based on user responses, such as the initiating organization, contract status, and preferred deployment date, the urgency level is categorized as:
  - *Urgent:* Deployment required within 3 days.
  - *High:* Deployment required within 10 days.
  - *Normal:* Deployment acceptable in more than 10 days.

This classification directly influences the scheduling and prioritization of technical and operational resources.

- **Infrastructure:** The user is prompted, to specify infrastructure preferences, which guide the decision between various cloud service models:
  - *Software as a Service (SaaS)* – minimal configuration, fast deployment, limited control.
  - *Platform as a Service (PaaS)* – some configurability, balanced control and automation.
  - *Infrastructure as a Service (IaaS)* – full control of VMs or containers.
  - *Serverless* – event-driven, highly abstracted infrastructure for elastic workloads.

This decision determines the level of control, customization, and setup complexity the user is prepared to manage.

- **Location:** Deployment location is selected based on regulatory, compliance, or latency requirements. Options typically include:

- *On-Premises*: For sensitive data or controlled environments.
- *Public Cloud*: For broader scalability and global availability.

The choice informs security models and data residency considerations.

- **Resources:** In this section, users identify, the dominant performance requirements of the application. Prioritization may include:

- *High CPU processing* (e.g., compute-intensive tasks)
- *Memory optimization* (e.g., in-memory caching)
- *Storage capacity* (e.g., data warehousing or backups)

These inputs help allocate virtual resources optimised during deployment planning.

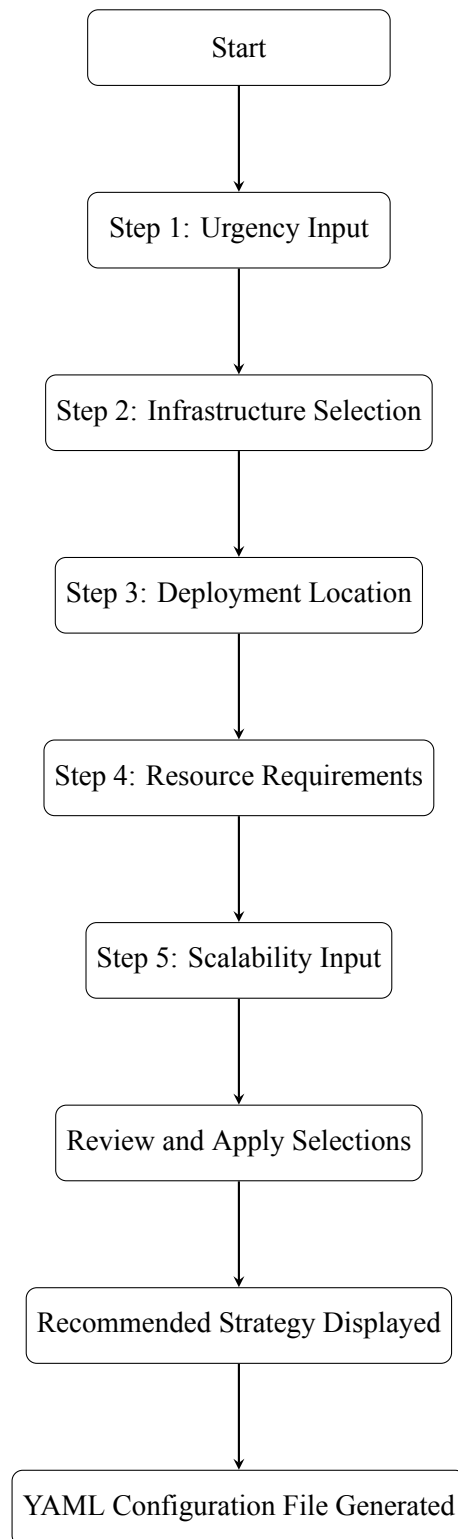
- **Scalability:** Scalability needs are specified, based on anticipated traffic patterns and elasticity:

- *Fixed Allocation*: Suitable for predictable, stable workloads.
- *Auto-scaling*: Necessary for applications with variable or bursty traffic.

This input impacts both cost modeling and performance guarantees.

- **Review and Final Decision:** After all inputs are gathered, a summarized review is presented to the user. The system uses internal reasoning to generate multiple deployment configurations, each justified by the answers provided. This enhances transparency, allowing users to make an informed final decision with full contextual feedback.

## Navigation Workflow Diagram



This structured and modular approach, not only guides the user intuitively through the configuration process, but also ensures accurate and complete data collection for system deployment and optimization.

### 3.3.2 User Interface Views

Figures 3.8–3.15, illustrate the user interface design, for each step of the process management workflow. Each figure presents, a dedicated configuration form developed with Thymeleaf, aimed at simplifying the input process and aligning each response with a specific deployment decision logic.

The screenshot shows a web form titled 'Which organization is submitting this request?'. It contains two radio button options: 'Ministry of Digital Governance' (with a brief description) and 'Other Public Sector Entity' (with a brief description). Below this is another question: 'Does your agency have an existing contract requiring immediate deployment?'. It has two radio button options: 'Yes' (with a brief description) and 'No' (partially visible).

Figure 3.8: Urgency step – collecting deployment urgency and organizational context.

The screenshot shows a progress bar at the top with seven steps: Start, Urgency, Infrastructure (current step), Location, Resources, Scalability, and Review. The 'Infrastructure' step is highlighted with a blue circle and number 3. Below the progress bar is a form titled 'Infrastructure Requirements'. It asks 'What is your primary objective?' and has three radio button options: 'Full Control & Customization' (with a brief description), 'Minimal Setup' (with a brief description), and 'Event-Driven Computing' (with a brief description).

Figure 3.9: Infrastructure preferences – selecting appropriate infrastructure model types (e.g., SaaS, PaaS, IaaS, Serverless).

**What type of data will this system handle?**

Personal Data (e.g., user accounts, customer details)

Selecting "Critical" may require additional security measures.

**Does this system need to connect to A.A.D.E. infrastructure?**

**Yes, this system requires an A.A.D.E. connection**  
A.A.D.E. infrastructure is used for tax and financial system integration.

**What is the budget allocation for this project?**

Low Budget (Cost efficiency, shared resources)

**Expected latency performance?**

Strict (Low Latency, real-time response)

**Scalability Requirements?**

Figure 3.10: Location selection – defining regional and compliance requirements for on-premises or public cloud deployment.

**Resource Priorities**

**What does your application prioritize?**

**Compute-Intensive**  
High CPU & RAM for fast processing, ideal for computation-heavy applications

**Storage-Intensive**  
Large storage capacity with lower CPU/RAM requirements, ideal for data-heavy applications


**What is the performance requirement for data access?**


**Low Latency**  
Fast response time for real-time applications and interactive services


**High Throughput**  
Batch processing, analytics, and high-volume data transfer capabilities


Figure 3.11: Resource configuration – prioritizing compute, memory, storage, and latency needs.


### Performance Parameters

 What is the expected average workload or user traffic for your application?

 **Low Traffic**  
 Minimal user base or workload, suitable for internal applications or early-stage projects

 **Medium Traffic**  
 Moderate user base or workload, common for business applications and established services

 **High Traffic**  
 Heavy user base or workload, for consumer-facing applications or high-throughput systems

 How frequently does your application experience spikes in workload or traffic?




 **Rarely**  
 Infrequent spikes in traffic or workload, predictable usage patterns

Figure 3.12: Performance parameters – defining expected traffic volume, spikes, and scalability behavior.


### Deployment Decisions Summary

 Infrastructure:

☐ Serverless (Azure Functions, etc.)
 ☐ Platform As A Service

 Scalability & Performance:

☒ Fixed Allocation: Suitable for predictable workloads with static resource limits. Auto-selected

 Urgency:

☒ Normal Priority: Regular deployment timeline. Standard resource allocation appropriate. All normal procedures and checks to be followed without modification. Auto-selected

Figure 3.13: Final decision interface – the user reviews the collected inputs and selects between appropriate decisions

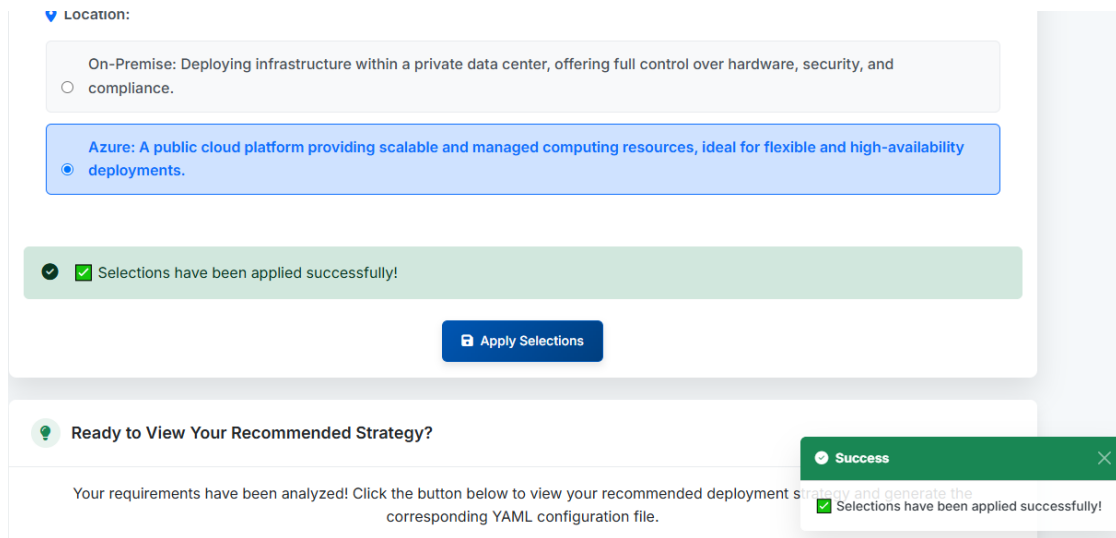


Figure 3.14: Final selection submission – the user applies the selected deployment configuration. This action triggers the backend processing engine to finalize the decision logic and prepare for strategy execution.

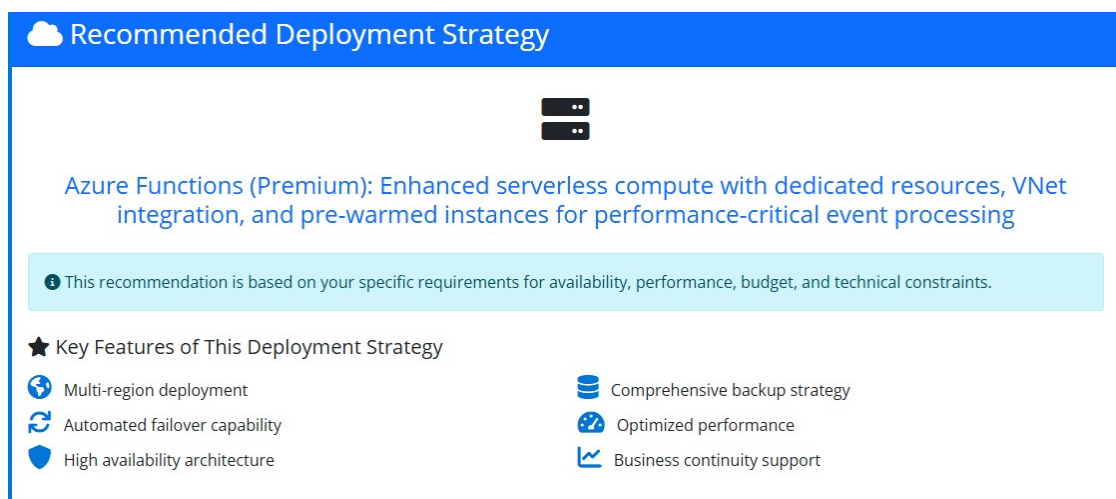


Figure 3.15: System’s final decision output – this view presents the recommended deployment strategy derived from the user’s structured inputs, supported by reasoning and explanation generated by the argumentation engine.

Following the system’s recommendation, users are granted access, to download a pre-generated YAML configuration file. This file, encapsulates, the selected deployment parameters, in an infrastructure-as-code format, enabling seamless provisioning and integration with modern DevOps pipelines.

## 3.4 Back-end Logic and Processing

Coordinating the data transformation and reasoning process, depends on the back-end of the system. It acts as a middle, between the user-interface and the Gorgias Cloud thinking engine.

Developed with Spring Boot Framework, the back-end service evaluates and organises the data, into logical facts following user interface organised inputs. These dynamically combined elements, fit the required syntax of the Prolog-based argumentation engine. Based on the situation, being handled, e.g., urgency, infrastructure, scalability, an appropriate Prolog query (e.g., `urgency(X)`) is built and sent over a RESTful API to the pertinent Gorgias Cloud project, together with the produced facts.

The Gorgias engine, compares the facts to the policy guidelines, stated in its hosted `.pl` file asynchronously processing every query. Usually in a structured JSON format, the response consists of both the conclusion of the inference (e.g., `X = high`) and a traceable justification of the reasoning route.

After receipt, the answer is broken out by the back end, which then maps the relevant conclusions to human-readable explanations. These outputs are then returned back to the front end, where they are displayed as clear summaries and useful recommendations, as seen in Figure 3.3. Table 3.1 summarizes the main logical components of the back-end system and their respective roles in the data processing pipeline.

Component	Role
Input Processor	Handles user inputs from forms and converts them into structured data
Fact Generator	Transforms structured input into logical facts suitable for reasoning (e.g., Prolog-style)
Query Dispatcher	Sends POST requests to the reasoning engine with both facts and relevant queries
Response Handler	Receives responses and interprets the output from the reasoning engine
Explanation Mapper	Maps raw reasoning results into user-friendly explanations and highlights important justifications

Table 3.1: Responsibilities of system components

### 3.4.1 UI Data Submission

Once a user completes the web-based decision form, the input is submitted via a POST request to the `/workflow/process-urgency` endpoint, in the case of urgency, using a Thymeleaf-annotated HTML form (see Code 3.6). Upon receipt, the Spring Boot backend stores the structured values in an embedded H2 database, enabling auditing and future traceability.

Immediately after persistence, the system initiates the logic processing pipeline. The stored values are programmatically transformed into Prolog-style facts relevant to the decision domain (as shown already in Code 3.2). These facts, along with the appropriate logic



queries (e.g., the urgency classification query in Code 3.3), are dispatched to the Gorgias Cloud reasoning engine for evaluation.

This modular flow ensures a clean separation between data collection, fact generation, and logical reasoning, supporting all five use case scenarios with consistency and reusability.

```
<form th:action="@{/workflow/process-urgency}" th:object="${form}"
    method="post">

    <!-- Organization Selection -->
    <div class="form-group">
        <label>Which organization is submitting this request?</label>

        <label class="custom-radio-card" for="opsRequest">
            <input type="radio" th:field="*{requestType}" id="opsRequest" name="requestType" value="ops"
                onclick="selectCard(this)">
            <div class="custom-radio-icon">
                <i class="fas fa-building"></i>
            </div>
            <div class="custom-radio-content">
                <div class="custom-radio-title">Ministry of Digital Governance</div>
                <p class="custom-radio-description">Other Public Sector or Entity</p>
            </div>
        </label>
    </div>
</form>
```

Code 3.6: User POST form request to initiate urgency classification

### 3.4.2 Dispatching the Gorgias Query via a Swagger-Generated Client

Derived from the Gorgias Cloud OpenAPI description, we use a statically typed Java client to control the way the system generates and sends HTTP requests to the Gorgias Cloud. Built on officially specified OpenAPI 2.0 (Swagger) schema supplied by the service, this client was created using the Swagger Codegen tooling. Together with a range of highly-typed request and response classes, such as *Gorgias Query* and *Gorgias Query Result*, the generated module was integrated into the Spring Boot application and gives a specific controller interface for launching reasoning queries. This strategy offers multiple benefits. First, it guarantees type safety, by using produced Java classes to enforce HTTP request and response payload structure at build time. Second, by abstracting low-level issues, including JSON serialisation, deserialisation, and error handling, it drastically reduces boilerplate code. At last, it makes maintainability feasible, since the client might be generated from an updated OpenAPI specification, therefore allowing perfect integration of any forthcoming improvements to the Gorgias Cloud API.

Code 3.7 shows the core method that prepares a Gorgias query and sends it to the remote engine. The method collects logical facts from the current form, configures the rule file(s) to be used during evaluation, and dispatches a POST request to the `/GorgiasQuery` endpoint.

```

public List<ParsedResult> executeGorgiasQueryForUrgency(
    WorkflowForm form, HttpSession session) {

    GorgiasQuery gorgiasQuery = new GorgiasQuery();
    setupGorgiasFiles(gorgiasQuery); // Specify the rulebase

    List<String> facts = collectFacts(form);
    if (facts.isEmpty()) {
        return Collections.emptyList();
    }

    gorgiasQuery.setFacts(facts);
    gorgiasQuery.setResultSize(5);
    gorgiasQuery.setQuery("urgency(X)");

    return performQuery(gorgiasQuery); // Uses generated API
}

```

Code 3.7: Submitting facts and a query to Gorgias Cloud via Swagger-generated client

The API call returns a *GorgiasQueryResult*, which includes metadata flags and a list of query results. Each result contains a set of variable bindings, rule justifications, and a natural-language explanation.

This mechanism decouples the reasoning backend from the application logic, while preserving transparency, extensibility, and testability across use cases.

### 3.4.3 Post-processing of Gorgias Results

The interface with *Gorgias Cloud* returns a JSON structure (*GorgiasQueryResult*) containing:

- **Solution** – eg. the final urgency level (normal, high, urgent);
- **Supporting Facts** – the set of facts supporting the decision.

However, this format is not immediately understandable to non-technical users. To present both the urgency level and the underlying justification in a **clear and human-friendly way**, we implemented a post-processing layer in **Java** that:

1. decodes the raw result,
2. maps technical fact identifiers to natural language descriptions (e.g., `request_type(other)` → “The request is from another ministry or organization”),
3. produces a list of `ParsedResult` items ready for UI display.

```

/** Converts GorgiasQueryResult to human-readable form. */
public List<ParsedResult> parseGorgiasQueryResult(
    GorgiasQueryResult gorgiasQueryResult) {

    Map<String, String> factMappings = new HashMap<>();
    populateMappings(factMappings);

    return gorgiasQueryResult.acceptedFacts().stream()
        .map(fact -> new ParsedResult(
            mapPriority(
                gorgiasQueryResult.classification()),
            factMappings.getDefault(fact, fact)))
        .toList();
}

```

Code 3.8: Extracting and transforming the Gorgias result

```

/** Creates a dictionary from technical IDs to descriptions. */
public void populateMappings(Map<String, String> factMappings) {

    factMappings.put("request_type(ops)",
        "The request is from the Ministry of Digital Governance.");

    factMappings.put("request_type(other)",
        "The request is from another ministry or organization.");

    // ...more mappings can be added as needed
}

```

Code 3.9: Mapping internal fact IDs to human-friendly labels

**Function mapPriority.** This helper method converts the raw classification string (e.g., "**high**") into a display-friendly version, such as “High Priority”, ready for UI output.

## Example Output

The resulting list appears in the UI as follows:

### High Priority

- └─ The request is from another Ministry or organization.
- └─ There is no current contract with any external contractor..

In this way, the final user receives a clear urgency **classification** (Normal / High / Urgent), with natural language justifying the decision.

Similar architecture, allows the post-processing layer to be applied to the final suggested deployment plan. For instance, internal result identifiers, like `azure_vm` map user-friendly names, like “Deploy on Microsoft Azure – Virtual Machine”. This guarantees that the urgency categorisation and the proposed technological solution are presented in a way that non-technical stakeholders can grasp, therefore preserving the main purpose of explainability and transparency of the system.

### 3.5 Argumentation Engine Design

Following the transformation of user input into logical facts, the system proceeds, to execute the argumentation rules, that drive the decision-making process. These rules were modeled using Raison AI, which translates input scenarios into formal Prolog .pl files, capturing complex policy logic in a structured and maintainable format. The rule files are hosted on Gorgias Cloud, which is integrated into the system proposed in this thesis via its API for remote execution. Upon receiving the input facts, Gorgias Cloud evaluates them against the defined logic, applies preference-based reasoning to resolve conflicts and returns transparent, explainable conclusions, that inform the final deployment recommendations.

#### Demonstration of Argumentation Logic: Urgency Case

To illustrate, the structure of policy reasoning, implemented in the argumentation engine, a subset of the decision rules, encoded in Prolog is presented in Code 3.10. Each Prolog .pl file consists of two main components: a set of logical *rules* that define the decision logic, and a set of *facts* that represent the specific input conditions.

For the reasoning engine, to produce a valid and context-aware conclusion, both elements must be present, rules to guide the logic and facts to represent the user input scenario. In addition, a prompt (i.e., a query) must be submitted, to instruct the engine on what to evaluate. These rules, in the case of urgency, determine the appropriate urgency level and establish preferences between alternative outcomes, based on context-specific criteria, such as agency type, contractor involvement, and timing constraints.

```
:- dynamic contract_with_contractor/1,
           highbasedondate/0, normalbasedondate/0,
           request_type/1, urgentbasedondate/0.

rule(d1, urgency(high), []) :-
    request_type(ops),
    contract_with_contractor(no),
    highbasedondate.

rule(d2, prefer(d1, r1), []) :-
    request_type(ops),
    contract_with_contractor(no),
    highbasedondate.

rule(r1, urgency(urgent), []) :-
    request_type(ops).

rule(r9, urgency(normal), []) :-
    contract_with_contractor(no).

complement(urgency(high), urgency(normal)).
complement(urgency(high), urgency(urgent)).
```

Code 3.10: Sample Argumentation Rules in Prolog

We used the Raison AI interface, which offers a simple graphical environment for scenario description, rule creation, and conflict resolution, to help us model and build these argumentation rules. Important phases in the configuration process, such as scenario development, option modelling, conflict resolution, and runtime query execution, are shown in Figures 3.16–3.21. Still exporting accurate .pl files for the reasoning engine, this tool enables users visually design complex decision structures, without hand Prolog coding. Its visual transparency guarantees accuracy and usability even for experts in domains other than formal logic.

urgency24022025

Initial scenarios and options > Compatible scenarios > Conflict resolution > Decision policy

NATURAL LANGUAGE

ADD A SCENARIO	MANAGE ELEMENTS	ADD AN OPTION	urgency(normal)	urgency(high)	urgency(urgent)
request_type(mog)	contract(no)	urgentBasedOnDate	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
agency_category(independentAuthority)	contract(no)	urgentBasedOnDate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
agency_category(localGovernment)	contract(no)	urgentBasedOnDate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
request_type(other)			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
highBasedOnDate			<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
urgentBasedOnDate			<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
contract(no)			<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
request_type(mog)			<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
agency_category(localGovernment)			<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 3.16: Scenario modeling using Raison AI

urgency24022025

Initial scenarios and options > Compatible scenarios > Conflict resolution > Decision policy

NATURAL LANGUAGE

Select a scenario to resolve a conflict

8 / 8 CONFLICTS SOLVED

solved ✓

agency\_category(independentAuthority) contract(no) urgentBasedOnDate

solved ✓

contract(no) agency\_category(localGovernment) urgentBasedOnDate

solved ✓

contract(no) agency\_category(localGovernment) urgentBasedOnDate request\_type(other)

solved ✓

agency\_category(independentAuthority) contract(no) urgentBasedOnDate request\_type(other)

Figure 3.17: Conflict resolution interface (8/8 resolved)

urgency24022025

Run application

NATURAL LANGUAGE

Choose input data

Data

highBasedOnDate contract(no) request\_type(other)

RUN

Results

urgency(normal) urgency(high) urgency(urgent)

Figure 3.18: Project execution to test logic

+	ADD A SCENARIO	+	MANAGE ELEMENTS	+	ADD AN OPTION	serverless	saas	paas	iaas
	consumeReadySoftware		noControl			<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	custom_integrations					<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	low_budget					<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	high_budget		data_sensitivity			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	runCustomApps		fullControl			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	eventDrivenFunctions					<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	consumeReadySoftware		noControl		high_budget	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	consumeReadySoftware		limitedControl			<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
	custom_integrations								

Figure 3.19: Scenarios for infrastructure

+	ADD A SCENARIO	+	MANAGE ELEMENTS	+	ADD AN OPTION	fixed_allocation	auto_scaling
	expectedLoad(high)					<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	expectedLoad(low)					<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	expectedLoad(medium)					<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	costSensitivity(high)					<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	costSensitivity(low)					<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	responseTime(medium)					<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	responseTime(high)					<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
	peakTimes(frequently)					<input type="checkbox"/>	<input checked="" type="checkbox"/>
	peakTimes(always)					<input type="checkbox"/>	<input checked="" type="checkbox"/>
	expectedLoad(low)		costSensitivity(low)				

Figure 3.20: Decision matrix for scalability

+	ADD A SCENARIO	+	MANAGE ELEMENTS	+	ADD AN OPTION	propose_location(azure)	propose_location(onPremise)
	nonpersonal					<input checked="" type="checkbox"/>	<input type="checkbox"/>
	personal					<input type="checkbox"/>	<input checked="" type="checkbox"/>
	critical					<input type="checkbox"/>	<input checked="" type="checkbox"/>
	tax					<input type="checkbox"/>	<input checked="" type="checkbox"/>
	medical					<input type="checkbox"/>	<input checked="" type="checkbox"/>
	nonpersonal		onPremise		high_budget	<input checked="" type="checkbox"/>	<input type="checkbox"/>
	medium_scalability		high_disaster_needs		general	<input type="checkbox"/>	<input type="checkbox"/>
	high_disaster_needs		general		high_scalability	<input checked="" type="checkbox"/>	<input type="checkbox"/>
	flexible_latency					<input type="checkbox"/>	<input checked="" type="checkbox"/>
	azure		personal			<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 3.21: Scenario and outcome mapping for location policy

Preference rules, are especially important for resolving conflicts, when multiple rules apply simultaneously.

## Rule Set Size

The Gorgias Cloud maintains an extensible and scalable set of rules, which can vary depending on the scenario complexity. For our system, we developed a total of six distinct reasoning projects, using Raison AI, each exported as a separate .p1 file. These files are hosted and executed remotely, through the Gorgias Cloud platform via API integration.

Each .p1 file corresponds to a specific decision-making dimension, including five case-specific modules, *Urgency*, *Infrastructure*, *Location*, *Resources*, and *Scalability*, as well as one additional module, for the *Final Decision*. Each file contains approximately 80–100 logical rules, capturing the full range of policy scenarios, relevant to that dimension. Exception is the final decision, .p1 (100-200 rules), which evaluates all the previous decisions, including user produced facts.

The final decision module, synthesizes the outputs of the five individual cases, enabling the system, to generate context-aware and preference-driven deployment recommendations. This modular rule architecture, supports both granular policy evaluation and holistic strategy formulation.

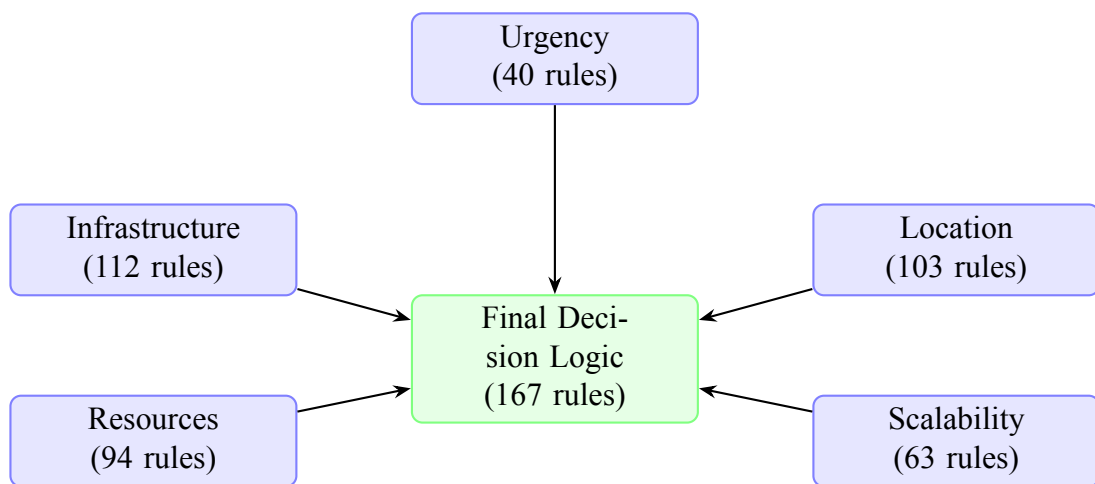


Figure 3.22: Modular Prolog rule sets hosted and executed in Gorgias Cloud



# Chapter 4

## Implementation

This chapter addresses, the technical setup of the development environment and the technical execution of the cloud deployment decision support system. The development approach took advantage of several modern technologies and frameworks in order to assure modularity, scalability, and maintainability. **Visual Studio Code (VS Code)** provided efficient code navigation, debugging, and **Git** integration, therefore serving as the main integrated programming environment (IDE). The back-end was developed using the **Spring Boot** framework, which gave a solid basis for building RESTful services and controlling logic orchestration. An embedded **H2** database was used to provide user-submitted data and intermediate states lightweight, in-memory storage throughout the development and testing phases. To enable integration with the Gorgias Cloud platform, the API documentation, was investigated and tested using **Swagger Editor**. This guaranteed correct formatting and organisation of the logical searches and answers. These components interacted to provide a harmonic surroundings that enabled the application of the system covered in this chapter.

### 4.1 Technology Stack and Tools

To build the cloud deployment decision support system, a carefully selected set of tools and technologies was utilised, each chosen for its compatibility, performance, and ease of integration. Figure 4.1 provides an overview of the system's development environment and technology layers.

- **VS Code:** Used as the primary Integrated Development Environment (IDE), offering efficient navigation, plugin support for Java and Spring Boot, REST API testing, and seamless Git integration for version control.
- **Git:** Employed for source code version control and collaboration, enabling structured branching, tracking of changes, and integration with GitHub for remote repository management.
- **Spring Boot:** Formed the core of the back-end application, supporting RESTful API design, modular service development, and dynamic fact generation for logical inference.
- **Maven:** Used for dependency management, and reproducible configuration across development and production environments.

- **H2 Database:** Provided an in-memory relational data store used during development for storing user submissions, state transitions, and audit records.
- **Thymeleaf + HTML/CSS:** Enabled dynamic front-end rendering of decision forms and configuration views, closely coupled with Spring controller logic.
- **Swagger Editor:** Utilized for interacting with and validating the Gorgias Cloud RESTful API, ensuring that logic queries were correctly formed and executed.
- **Docker:** Used to containerize the application for deployment testing, offering consistency across environments and simplifying dependency setup and portability.
- **Gorgias Cloud + Raison AI:** Served as the external logic engine and rule modeling environment. Raison AI facilitated rule creation via graphical modeling, while Gorgias Cloud evaluated the exported .p1 files against submitted facts.

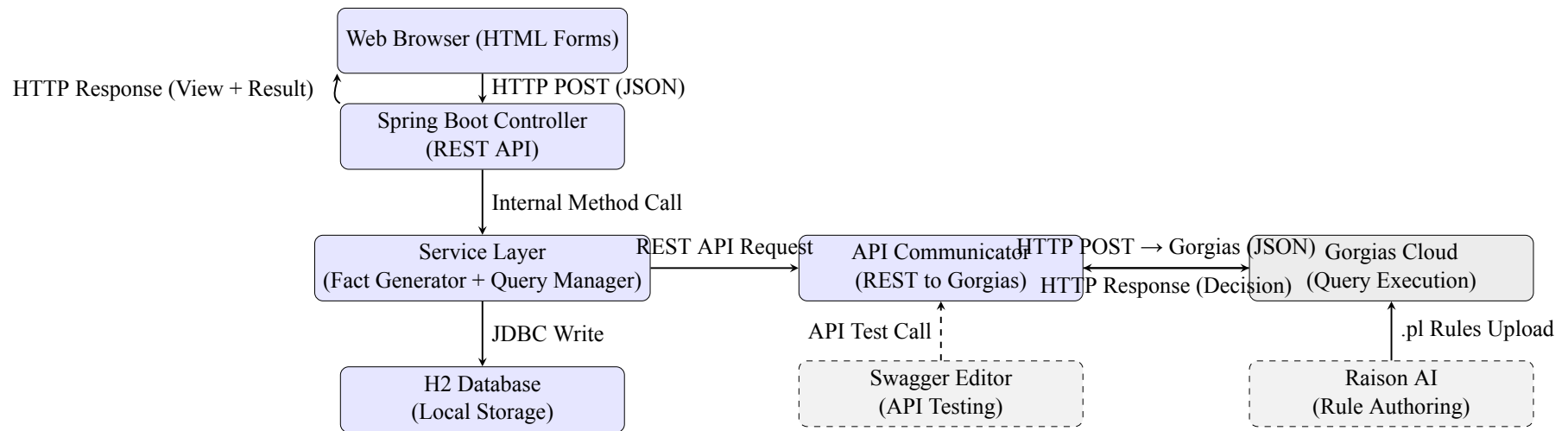


Figure 4.1: Enhanced implementation architecture with RESTful requests and responses, internal Spring Boot logic, and integration with external reasoning and rule tools.

## 4.2 Development Environment and Project Setup

To implement the deployment recommendation system, a lightweight and modular development environment was chosen to maximize flexibility and automation. The core technologies include Visual Studio Code as the development IDE, Spring Boot for the application framework, and Maven for build management. The project is structured using a RESTful service-oriented architecture, with integrated reasoning capabilities via the Gorgias Cloud API.

### 4.2.1 VS Code and Spring Boot Setup

The project was developed using **Visual Studio Code (VS Code)**, which provides robust Java support through extensions, such as Spring Boot Tools, Debugger for Java, and Lombok Annotations. The VS Code environment supports integrated terminal access, Git operations, and real-time Maven build status.

Project scaffolding was bootstrapped using the **Spring Initializr** service, with the following base configuration:

- **Language:** Java 17
- **Build Tool:** Maven
- **Dependencies:**
  - spring-boot-starter-web
  - spring-boot-starter-thymeleaf
  - spring-boot-starter-data-jpa
  - com.h2database:h2
  - swagger-codegen-maven-plugin

### 4.2.2 Dependency Management: pom.xml

The `pom.xml` file manages all build and runtime dependencies, allowing for reproducible builds and smooth integration. Code 4.1 displays a snippet of the configuration for Swagger and H2.

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>io.swagger</groupId>
  <artifactId>swagger-codegen</artifactId>
  <version>3.0.23</version>
</dependency>

```

Code 4.1: Sample dependencies in pom.xml

The Swagger dependency enables automatic generation of client-side Java code to interact with the Gorgias Cloud REST API. This simplifies query construction and ensures endpoint compatibility.

### 4.2.3 Integration of Gorgias Cloud API via Swagger Codegen

To enable structured and type-safe interaction with the Gorgias Cloud reasoning service, a Java client was generated directly from its OpenAPI (Swagger) specification. Rather than consuming a precompiled client library, the `swagger-codegen-maven-plugin` was used to generate Java source code, based on the official `gorgias-swagger.json` file provided by Gorgias.

This OpenAPI specification formally describes the available endpoints, including `/GorgiasQuery`, as well as their expected input and output structures. This ensures that the client code is schema-compliant and aligned with the API contract.

The plugin was configured in the project's `pom.xml` file as shown in Code 4.2:

```

<plugin>
  <groupId>io.swagger.codegen.v3</groupId>
  <artifactId>swagger-codegen-maven-plugin</artifactId>
  <version>3.0.23</version>
  <executions>
    <execution>
      <id>generate-gorgias-client</id>
      <goals>
        <goal>generate</goal>
      </goals>
      <configuration>
        <inputSpec>${project.basedir}/src/main/resources/gorgias-swagger.json</inputSpec>
        <language>java</language>
        <output>${project.build.directory}/generated-sources/swagger</output>
        <apiPackage>com.myapp.gorgias.api</apiPackage>
        <modelPackage>com.myapp.gorgias.model</modelPackage>
      </configuration>
    </execution>
  </executions>
</plugin>

```

Code 4.2: Swagger Codegen plugin in pom.xml

The client classes were generated by running:

```
mvn swagger-codegen:generate
```

This command produced Java code that was imported directly into the project. The generated classes, such as API controllers and data models, are then used within service layers to construct and send requests to Gorgias Cloud and to process the reasoning results in a structured way.

A representative API usage pattern involves invoking the /GorgiasQuery endpoint, which receives a logical query and returns a reasoned response, as shown in Code 4.3

```

1 POST /GorgiasQuery
2 Content-Type: application/json
3
4 {
5   "facts": ["request_type(other)", "highBasedOnDate"],
6   "query": "urgency(X)",
7   "resultSize": 1
8 }

```

Code 4.3: Example API Call Body

The Java application, constructs this payload, using the generated model classes, invokes the API using the client stub, and extracts the response, typically including the decision, an explanation, and any variable bindings.

**Note:** There is no external JAR or precompiled library labeled "Gorgias API client." The integration is entirely achieved through Swagger-based code generation, making the interaction with the reasoning service consistent, maintainable, and aligned with schema definitions.

#### 4.2.4 Version Control with Git

Git was used as the primary version control system to manage source code, track development history, and support collaboration workflows. The project followed a structured branching strategy to distinguish between stable releases, development work, and experimental features.

The repository structure included:

- `main/` — stable production-ready commits
- `dev/` — main development branch
- `feature/{name}` — dedicated branches for isolated modules or improvements

Development was staged across 12 tagged versions, named sequentially from `michalakis-v1` to `michalakis-v12`. These tags reflected major milestones or iterations of the system, such as UI overhauls, logic integration, and external API connectivity.

The Git commands in Code 4.4 were frequently used to manage the repository:

```
# Create a new feature branch
git checkout -b feature/urgency-module

# Stage and commit changes with a structured message
git add .
git commit -m "feat(api): add urgency classifier query logic"

# Push the branch to the remote repository
git push origin feature/urgency-module

# Rename a branch for versioning
git branch -m michalakis-v12
```

Code 4.4: Typical Git commands used during development

The repository was hosted privately on GitHub, supporting collaboration through pull requests, code reviews, and issue tracking. Although CI/CD was not fully configured, the project structure supports future integration via GitHub Actions or Jenkins pipelines.

#### 4.2.5 Containerization with Docker

Docker was utilized to containerize the application environment, enabling consistent execution across different machines and deployment targets. The primary objective was to package the Spring Boot application and its dependencies (including the H2 database and external API communication) into a portable and reproducible container image.

A custom Dockerfile was defined in the project root, shown in Code 4.5

After building the Spring Boot application, the Docker image was created using the commands in Code 4.6

```
# Base image with JDK 17
FROM openjdk:17-jdk-slim

# Add application JAR file
COPY target/michalakis-v12.jar app.jar

# Expose default port
EXPOSE 8080

# Command to run the app
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Code 4.5: Sample Dockerfile used for Spring Boot containerization

```
# Build the image
docker build -t michalakis-app:v12 .

# Run the container on port 8080
docker run -p 8080:8080 michalakis-app:v12

# Push the image to Docker Hub (for cloud deployment)
docker tag michalakis-app:v12 <your-dockerhub-username>/michalakis-app:v12
docker push <your-dockerhub-username>/michalakis-app:v12
```

Code 4.6: Docker build and run commands

This setup made it easy to test the system locally in a clean environment, deploy it to a virtual server, and later integrate it into cloud environments as needed. By pushing the Docker image to Docker Hub, deployment to services, such as Render.com, was simplified, allowing the application to be launched directly from the cloud-hosted container image.

Docker also helped simulate realistic deployment conditions for communicating with the Gorgias Cloud API, as it ensured consistent runtime environments, even when switching between machines or network configurations.

## 4.3 Automated Configuration File Generation

To support the suggested deployment strategies, aligned with the Ministry of Digital Governance's decision framework, the system automatically generates YAML configuration files, tailored to Microsoft Azure Products.

Figure 4.2 (adapted from [32]) illustrates the official flowchart employed by the Ministry to guide service selection in Microsoft Azure (e.g., VM, Functions, AKS), based on factors like control, workload type, containerization capability, and orchestration preference.

### 4.3.1 How the Final Deployment Strategy Is Determined

The final deployment strategy is determined through a rule-based reasoning mechanism, consistent with the logic applied in all other stages of the system. This mechanism combines



## Choose a candidate service

Use the following flowchart to select a candidate compute service.

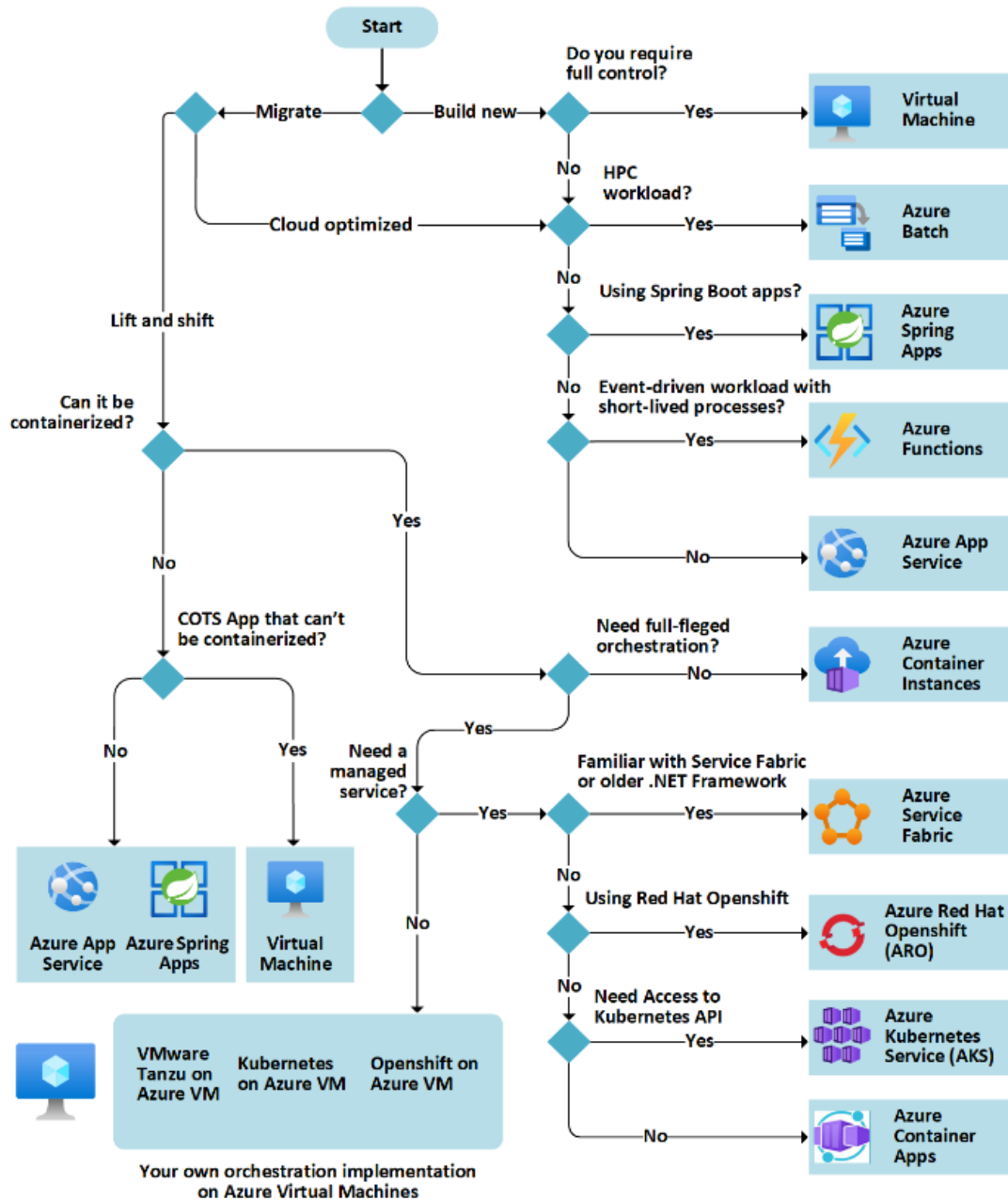


Figure 4.2: Azure service selection flowchart used by the Ministry of Digital Governance

prior case-based decisions with user-defined inputs to identify the most suitable deployment path. A predefined rule file is used, which encodes the outcomes of previous cases alongside user-selected parameters gathered during the workflow process.

The following code snippet in Code 4.7 is a representative rule derived from a combination of system-inferred decisions and user preferences:

```
rule(p59(azure_high_performance_compute),
    prefer(r13_1(azure_high_performance_compute),
        r1_5(azure_vm_high_performance)),
    [high_performance_compute]).
```

Code 4.7: Example Preference Rule with Case-Based and User-Defined Criteria

In this example, the system prefers the `azure_high_performance_compute` configuration over the more general `azure_vm_high_performance`, when the user explicitly requests high-performance computing capabilities. This preference reflects both the user's current input and strategic choices, derived from prior deployment cases.

As another example, consider a scenario in which the user specifies "high memory" as a primary requirement. If past cases indicate that, serverless architectures, offer better elasticity and cost-efficiency, under similar constraints, the system satisfies a rule that favors memory-optimized Azure Functions, as shown in Code 4.8

```
rule(p63(azure_function_memory_optimized),
    prefer(r17_2(azure_function_memory_optimized),
        r3_3(azure_vm_memory_optimized)),
    [high_memory]).
```

Code 4.8: Memory-Optimized Serverless Preference

This dynamic and adaptive approach, ensures that deployment recommendations are both context-aware and informed by accumulated experience, thereby enabling intelligent automation in complex cloud environments.

### 4.3.2 Official Flowchart Connection System Output

We reproduce the official Azure service, choosing flowchart using a logic-based decision approach (see Figure 4.2). After evaluating main user inputs and each case decision, such as the type of application (e.g., Serverless and Urgency timeline), it selects the best Azure service category.

If the user wishes total control over the virtualised environment, the system proposes ***Infrastructure as a Service (IaaS)***, most specifically Azure Virtual Machines. Typically, using Azure App Services or Azure Spring Apps, the system selects ***Platform as a Service (PaaS)*** for standard web apps lacking container functionality. Program driven events with brief runs call for ***Serverless*** computing using Azure Functions. When the workload is containerizable, the system recommends either container-oriented solution, Azure Container Instances (ACI) or Azure Kubernetes Service (AKS). At last, depending on the use case, the system may also offer ***Software as a Service (SaaS)*** alternatives for users that value low-management overhead and turn-key deployment.

Once the suitable category has been chosen, the system links this option to a pre-defined Azure deployment artifact, usually a templated YAML file. These models are made to fit every Azure service path, thereby enabling automation and consistency in keeping with Infrastructure-as- Code (IaC) best standards.

### 4.3.3 Sample YAML Template

As a proof of concept, a sample template for an *Azure Function App (Premium)* deployment is shown in Code 4.9. Different branches in the flowchart trigger different templates (e.g., VM, AKS, Spring Apps).

```
apiVersion: 2023-01-01
kind: AzureDeployment
metadata:
  name: premium-functions
  description: -Highperformance Azure Functions (Premium plan)
parameters:
  functionAppName:
    type: string
    default: func-premium
  location:
    type: string
    default: westeurope
functions:
  - name: ${functionAppName}
    plan: Premium
    runtime: java
    vnetIntegration: true
    preWarmedInstances: 2
    tags:
      environment: production
      workloadType: serverless-premium
```

Code 4.9: Sample Azure Configuration File Generated

Though Azure services are the main objective of the system, its templating method is made to be somewhat flexible. Along with serverless designs using Azure Functions, it may create deployment configurations for on-site virtual machines, such as those controlled using VMware Tanzu and container orchestration systems, like AKS, Tanzu, or OpenShift. Likewise, classic PaaS choices, including App Service and Azure Spring Apps are also supported. Every one of these setups depends on the same underlying logic-to-template mapping to guarantee that every decision path exactly matches a ready-to-deploy YAML file.

## Summary

In conclusion, by bridging formalized user inputs, the Ministry's deployment flowchart, and code-generated YAML templates, our system automates the final step of the deployment life-cycle. The generated configuration files are immediately usable in IaC pipelines or DevOps tools, reducing manual effort, configuration errors, and deployment time.

# Chapter 5

## Evaluation And Results

To evaluate our System, a survey was conducted using a questionnaire based on the System Usability Scale (SUS). The questionnaire included 10 questions assessing various aspects of the application's usability, with responses on a 5-point Likert scale (1: Strongly Disagree to 5: Strongly Agree). The survey participants represented diverse professional backgrounds, providing a comprehensive evaluation across different user groups.

### 5.1 System Usability Scale (SUS) Questionnaire

The following 10 questions comprised the System Usability Scale (SUS) questionnaire, each rated on a 5-point Likert scale (1: Strongly Disagree, 5: Strongly Agree):

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

*Note: According to the standard SUS methodology, odd-numbered items (Q1, Q3, Q5, Q7, Q9) are positively worded, while even-numbered items (Q2, Q4, Q6, Q8, Q10) are negatively worded. The interpretation of responses is adjusted accordingly.*

## Participant Demographics

The evaluation involved **seventeen (17)** participants from various professional sectors:

- 29.4% (5/17) worked in the public sector
- 41.2% (7/17) were employees of software companies
- 17.6% (3/17) were freelancers in IT
- 11.8% (2/17) did not specify their occupation

This diverse sample allowed us to gather insights from different perspectives, including varying levels of technical expertise and industry experience.

## 5.2 Analysis of Results Using Boxplot Diagrams

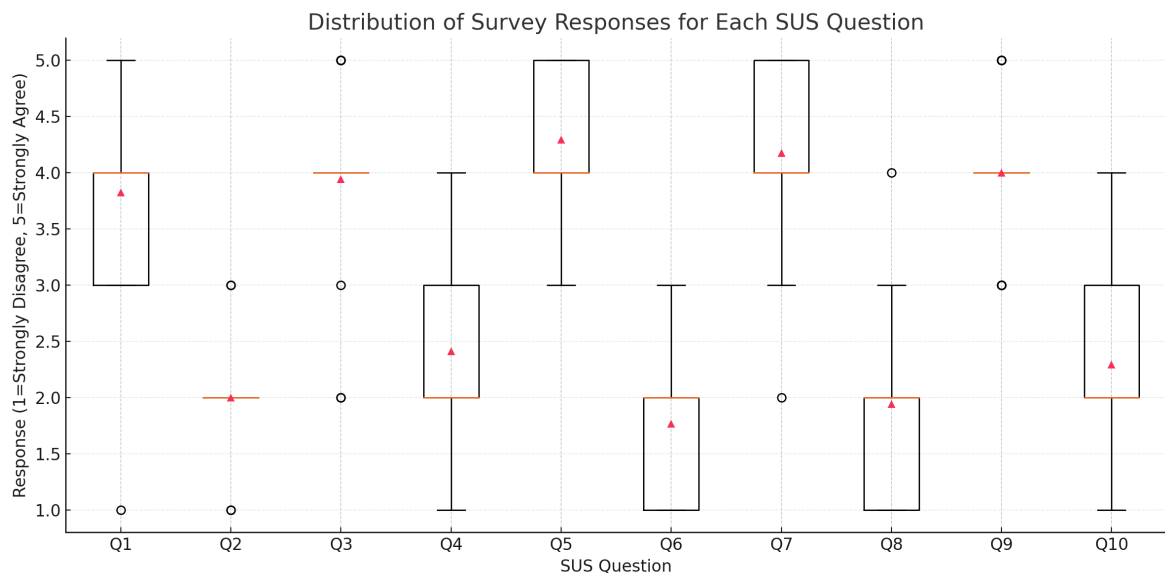


Figure 5.1: Distribution of participant responses for each System Usability Scale (SUS) question, as shown in boxplot diagrams. Each boxplot displays the median (red horizontal line), the interquartile range (IQR; box), and potential outliers (points beyond whiskers). The box represents the middle 50% of responses, while the whiskers indicate the full range, excluding outliers. This visualization enables rapid identification of consensus or disagreement among participants for each question. Higher medians and tighter boxes indicate greater consensus and satisfaction.

### Interpretation of Boxplot Results

The boxplots (Figure 5.1) reveal several key patterns in participant responses:

- **Questions with Positive Wording** (higher score = more positive evaluation): Q1, Q3, Q5, Q7, Q9.

- Most participants responded positively to these questions, indicating that the system is generally perceived as easy to use and well integrated. The median scores for these items were close to 1 or 2, suggesting a high level of satisfaction in these areas.
- **Questions with Negative Wording** (lower score = more positive evaluation): Q2, Q4, Q6, Q8, Q10.
  - The responses to negatively phrased questions are generally high (4 or 5), indicating that users did not perceive the system as complex, inconsistent, or difficult to learn. This further supports the conclusion that the system is user-friendly.

## Connection of SUS Questions to Evaluation Themes

Based on the content and intent of each SUS question, the responses were grouped into distinct evaluation themes, which reflect common usability dimensions. The analysis of the questionnaire responses allowed us to group SUS questions into specific themes, providing deeper insight into participant perceptions. The connection between SUS questions and evaluation themes is as follows:

- **Ease of Use:**
  - Q3 (ease of use), Q7 (quick learnability), Q8 (not cumbersome)
- **Consistency and Integration:**
  - Q5 (integration), Q6 (consistency)
- **User Confidence:**
  - Q9 (confidence), Q4 (minimal need for support)
- **Usage Intention:**
  - Q1 (frequent use), Q2 (not unnecessarily complex), Q10 (minimal learning required)

This explicit mapping clarifies how the evaluation themes emerged directly from the participants' responses, ensuring alignment between quantitative data and qualitative interpretations.

## System Usability Scale (SUS) Score Calculation and Interpretation

To quantify the overall usability of the application, we calculated the System Usability Scale (SUS) score for each participant based on their responses to the 10 SUS questions. Each question was scored according to the standard SUS methodology: for positively-worded questions (odd-numbered), the score is the response minus 1; for negatively-worded questions (even-numbered), the score is 5 minus the response. The total for each participant is then multiplied by 2.5, yielding a final SUS score ranging from 0 to 100 where higher numbers indicate better SUS score.

**Results:** The mean SUS score among all 17 participants was **74.6** (SD = 9.8), with individual scores ranging from 57.5 to 90.0. This mean is substantially higher than the industry

benchmark for average usability, which is set at 68. The distribution of SUS scores is shown in Figure 5.2.

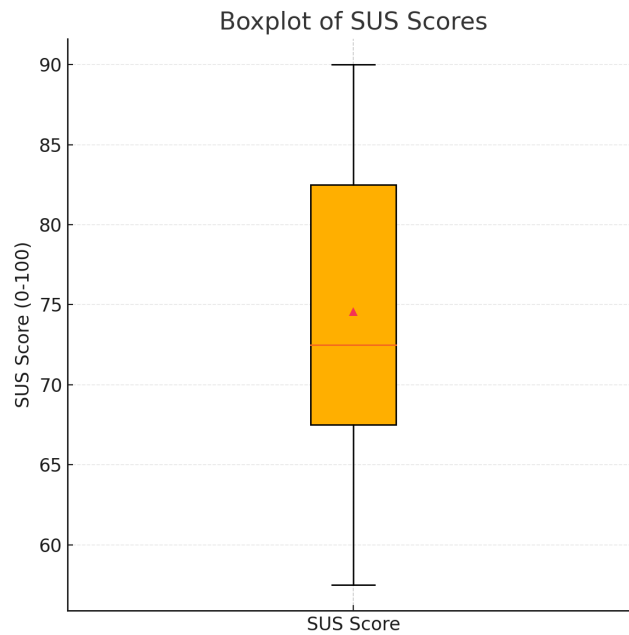


Figure 5.2: Distribution of calculated SUS scores across all participants.

According to standard benchmarks, SUS scores above 70 are considered "good" and scores above 85 indicate "excellent" usability. The mean SUS score of 74.6 suggests that users found the system to be generally usable and above average compared to typical applications. This indicates that, while the system meets a good standard of usability, there is still some room for improvement in the user experience. The distribution also shows variability among participants, with individual scores ranging from 57.5 to 90.0; a few users rated the system as excellent, while others provided lower scores, reflecting differing user experiences.

**Note on the Discrepancy:** Although qualitative feedback indicates generally positive usability impressions, some variability in SUS scores suggests that a subset of users faced specific barriers or confusions that affected their experience. Identifying these precise issues should be prioritized in future refinements to further improve overall usability.

Overall, the results highlight key areas for further development and refinement, to improve the overall usability and user satisfaction with the application.

## Practical Implications

The decision support system, offers significant benefits for organizations, seeking to optimize their cloud infrastructure deployment strategies. It reduces the complexities, involved in understanding and selecting among various deployment options, which is particularly beneficial for entities without in-depth technical knowledge of cloud architectures. However, the system's effectiveness is on the continuous updating and maintenance of its decision logic and integration capabilities to adapt to evolving cloud technologies and market conditions.

# Chapter 6

## Conclusions and Future Work

This thesis presented the design and implementation of a decision support system, aimed at optimizing cloud deployment strategies, through an advanced argumentation-based model. By leveraging contemporary web technologies, specifically the Gorgias Cloud and Raison AI, the project not only addressed the intricacies of complex decision-making processes, but also prioritized accessibility for both technical and non-technical stakeholders. The system features a comprehensive parameterization interface, fostering detailed customization and analysis, to effectively guide users toward optimised cloud infrastructure decisions.

### Key Achievements

The project realized several notable achievements in pursuit of its objectives. First, the system automated intricate decision-making processes, significantly reducing the need for manual intervention and minimizing the likelihood of human error. Additionally, the implementation of an intuitive frontend interface simplified the user experience, making advanced decision support functionalities readily accessible. Furthermore, the integration of the Gorgias Cloud facilitated the establishment of a robust and adaptive argumentation framework, allowing the system to accommodate a wide array of user requirements and preferences. Finally, substantial enhancements were made, to the backend architecture to bolster the system's scalability and security, as well as to ensure seamless integration with external APIs for real-time data processing. Collectively, these accomplishments have contributed to the creation of a sophisticated and practical tool for cloud deployment decision support.

### 6.1 Future Work and Enhancements

Although this thesis has introduced a comprehensive argumentation-based decision-support system, tailored for cloud deployment within public sector environments, several avenues remain open for further development and enhancement. These prospective improvements pertain not only to the refinement of decision-making mechanisms, but also to broadening the system's scope and enhancing user accessibility.

One particularly promising direction involves the integration of the system with cloud-native DevOps pipelines. By embedding the decision-support framework, directly into continuous integration and deployment (CI/CD) workflows, it would be possible to automate the provisioning and management of cloud resources seamlessly. For instance, the system's decision outputs, such as YAML configuration files, could be injected into platforms, like Azure



DevOps, AWS CodePipeline, or GitHub Actions. Such integration would enable the automatic execution of deployment recommendations, based on real-time user input, continuous deployment of infrastructure configurations, as well as dynamic monitoring and rollback capabilities, ensuring that infrastructure choices remain adaptable to shifting resource demands and compliance requirements.

Further, while the current system leverages static, predefined Prolog rules within Gorgias Cloud and Raison AI, future work could incorporate machine learning-driven rule adaptation. By employing machine learning models to analyze historical deployment data and user feedback, the system could dynamically refine its argumentation rules, identify emerging patterns in infrastructure preferences, and adapt to evolving regulatory standards. The adoption of reinforcement learning techniques, could allow the system to continuously optimize its decision-making process in response to actual deployment outcomes, thereby increasing the relevance and accuracy of its recommendations as cloud technologies evolve.

Moreover, the system's decision scope can be extended beyond the current focus on infrastructure selection. Future iterations might encompass a broader range of decisions, including the configuration of security policies, the incorporation of compliance requirements such as GDPR, FedRAMP, or ISO 27001, the implementation of cost-optimization strategies, and the support for containerized application deployments. This holistic approach would better align the system with the complex realities of cloud governance and infrastructure planning in modern organizations.

At present, the system generates YAML files for infrastructure automation. Expanding support to include a variety of configuration formats, such as Terraform configurations, Ansible playbooks, AWS CloudFormation templates, and Helm charts for Kubernetes, would make the solution more versatile and accessible across diverse DevOps environments. Enabling users to select their preferred configuration format would facilitate smoother integration and broader adoption.

Finally, enhancing the user experience through a personalized and adaptive user interface, represents another important direction for future work. By leveraging user analytics, the system could personalize recommendations based on past interactions, provide contextual guidance and explanations for each decision, and support interactive scenario simulations to help users better understand the trade-offs, involved in different deployment strategies. Such improvements, would not only increase the system's intuitiveness, but also ensure its accessibility to both technical and non-technical stakeholders, ultimately broadening its impact and practical value.

## **6.2 Final Thoughts**

Completing this project, has significantly enhanced my understanding and skills, in both the theoretical and practical aspects of building sophisticated decision support systems. The journey, has been transformative, providing invaluable insights between cloud architecture, web applications and decision-making automations.

Throughout this research, I have witnessed firsthand how an automated decision support system, can bridge the gap between technical complexity and deployment needs. The development process, challenged me to balance technical depth, with a user friendly design, a critical consideration when creating tools intended for diverse professional audiences. The positive evaluation from both technical experts and non-specialists, validates this balanced approach for such systems.

This project has reinforced the crucial role of argumentation in the future of IT infrastructure management and set a foundation for ongoing innovation in cloud technology deployments. As cloud environments continue to evolve, with increasing complexity and options, the need for intelligent decision support will only grow. The methods and frameworks, developed in this thesis, provide a scalable foundation, that can adapt to emerging cloud technologies, and deployment paradigms.

Beyond the technical achievements, this work has highlighted the importance of human centered design in technical tools. The integration of usability principles with sophisticated argumentation, represents a significant shift in how we approach infrastructure management tools, moving from purely technical considerations, to a more holistic view that acknowledges the human elements of technology adoption and use.

The experiences and insights gained throughout this project, lay a solid groundwork for my future endeavors, in the field of cloud computing and technology development. They have equipped me with not only technical proficiency, but also a deeper understanding of how to create meaningful technological solutions that address real-world challenges. As cloud computing continues to enrich vital evolution, the principles and approaches developed in this research, will remain relevant and adaptable to new contexts and requirements.

Looking forward, I envision extending this work to incorporate emerging technologies, such as edge computing paradigms, multi-cloud orchestration, and integration with AI-driven operational analytics. The foundations established through this research, provide a robust platform for such future innovations, potentially transforming the way organizations approach their cloud infrastructure decisions and management.

In conclusion, this project represents not just an academic exercise, but a meaningful contribution to the field of cloud computing, that I hope will inspire further research and practical applications in this rapidly evolving domain.

# Bibliography

- [1] N. K. Janjua, *A Defeasible Logic Programming-Based Framework to Support Argumentation in Semantic Web Applications*. Springer Theses, Springer International Publishing, 1st ed., 2014.
- [2] I. Michalakis, “Cloud deployment assistant (deployai app).” <https://github.com/imichalakis/michalakis-thesis-code.git> and <https://michalakis-thesis-v12.onrender.com/>, 2024. Accessed: Mar. 10, 2025.
- [3] A. C. Kakas, P. Moraitis, and N. I. Spanoudakis, “Gorgias: Applying argumentation,” *Argument & Computation*, vol. 10, no. 1, pp. 55–81, 2019. Published online December 6, 2018.
- [4] F. H. van Eemeren and R. Grootendorst, *A Systematic Theory of Argumentation: The pragma-Dialectical Approach*. Cambridge University Press, 2004.
- [5] N. I. Spanoudakis, E. Constantinou, A. Koumi, and A. C. Kakas, “Modeling data access legislation with gorgias,” in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE 2017)*, pp. 317–327, Springer, 2017.
- [6] E. Karali, A. C. Kakas, N. I. Spanoudakis, and E. C. Lupu, “Argumentation-based security for social good,” in *2017 AAAI Fall Symposium Series*, 2017.
- [7] F. Cloppet, P. Moraitis, and N. Vincent, “An agent-based system for printed/handwritten text discrimination,” in *International Conference on Principles and Practice of Multi-Agent Systems (PRIMA 2017)*, pp. 180–197, Springer, 2017.
- [8] N. Spanoudakis and P. Moraitis, “Engineering an agent-based system for product pricing automation,” *Engineering Intelligent Systems*, vol. 17, no. 2, p. 139, 2009.
- [9] K. Pendaraki and N. Spanoudakis, “Portfolio performance and risk-based assessment of the portrait tool,” *Operational Research*, vol. 15, no. 3, pp. 359–378, 2015.
- [10] P. M. Dung, “On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games,” *Artificial Intelligence*, vol. 77, no. 2, pp. 321–357, 1995.
- [11] W. F. Clocksin and C. S. Mellish, *Programming in Prolog: Using the ISO Standard*. Springer, 5th ed., 2003.
- [12] A. C. Kakas and P. Moraitis, “Argumentation based decision making for autonomous agents,” in *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003)*, pp. 883–890, 2003.

- [13] A. J. García and G. R. Simari, “Defeasible logic programming: An argumentative approach,” *Theory and Practice of Logic Programming*, vol. 4, no. 1-2, pp. 95–138, 2004.
- [14] G. L. Turnquist, *Learning Spring Boot 2.0: Simplify the development of production-grade applications using Spring Boot*. Packt Publishing Ltd, 2017.
- [15] J. C. Johnson, *Spring Microservices in Action*. Manning Publications, 2019.
- [16] M. Raible, “Modern java web development: Bootstrapping, spring boot, and angularjs.” Presentation, 2016.
- [17] M. Fazio, A. Puliafito, and M. Villari, “A comparative evaluation of spring boot, dropwizard, and wildfly swarm for microservices development,” *Journal of Computer Science and Technology*, vol. 16, pp. 101–110, 2016.
- [18] J. Soldani, D. A. Tamburri, and W.-J. van den Heuvel, “Microservices migration in industry: intentions, strategies, and challenges,” in *2018 IEEE International Conference on Software Architecture (ICSA)*, pp. 29–290, IEEE, 2018.
- [19] N. A. Chohan, M. A. Qureshi, A. Ullah, and M. Z. Aziz, “Java frameworks for restful web services: A comparative study,” *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 7, pp. 524–531, 2019.
- [20] A. Myers and D. Thomas, “Continuous integration and continuous deployment practices in java web application development,” *Journal of Software Engineering and Applications*, vol. 14, no. 4, pp. 175–183, 2021.
- [21] T. Project, *Thymeleaf Documentation*, 2023. Available: <https://www.thymeleaf.org/documentation.html>.
- [22] D. Fernández, *Thymeleaf: Modern Server-side Java Template Engine for Web and Standalone Environments*. Thymeleaf Project, 2013.
- [23] I. Pivotal Software, *Spring Boot Reference Documentation*, 2023. Available: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>.
- [24] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, 2014.
- [25] C. Boettiger, “An introduction to docker for reproducible research,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.
- [26] C. Pahl, “Containerization and the paas cloud,” *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.
- [27] A. Martins, M. Ahmed, and J. Bernardino, “Security analysis of docker containers,” in *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pp. 0524–0530, IEEE, 2019.
- [28] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
- [29] S. Chacon and B. Straub, *Pro Git*. Apress, 2014.

- [30] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc., 2012.
- [31] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promise and perils of mining git," *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR)*, pp. 1–10, 2010.
- [32] Microsoft Docs, "Azure architecture decision flowchart," 2023. Available at <https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/compute-decision-tree>.