International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 11 | Nov - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

TechManager: A Tech Stack Recommendation System for Software Projects

Project Guide : **Dr Dharmendra Roy**

Mohammad Akhil [1] | M. Sanjay [2] | N. Sai Tanav [3] | P. Manikanta [4]
Department of Computer Science and Engineering
Hyderabad Institute of Technology and Management (HITAM)
Hyderabad, Telangana, India

Abstract

Selecting an appropriate technology stack is one of the earliest and most consequential decisions in a software project, yet undergraduate students and small development teams often make this choice informally, relying on familiarity or prevailing trends rather than a structured assessment of project requirements. This paper presents TechManager, a rulebased decision support system implemented as a Flask web application that assists a user in obtaining a suitable frontend technology, backend technology, database, hosting platform and software architecture for a proposed project. The system accepts four inputs — project name, project type, team size and budget — and applies a set of predefined domain rules to generate a recommendation together with an estimated development complexity, a scalability indication, an estimated development time and cost, suggested team roles, a development roadmap, security recommendations and alternative technology options. Six project domains are currently supported: portfolio, education, hospital, banking, e-commerce and e-sports. Every generated recommendation is persisted in a SQLite database, enabling recommendation history, search/filter and deletion through the web interface. Unlike systems that rely on machine learning or statistical inference from historical project data, the TechManager recommendation engine is entirely rule-based, which makes its outputs transparent, explainable and straightforward to modify or extend. This paper describes the motivation, system architecture, methodology, rule-processing algorithm, implementation and a functional, test-case-based evaluation of TechManager, and discusses the advantages and limitations that follow from adopting a rule-based approach to technology-stack recommendation for student and small-team software projects.

Keywords— *technology stack selection; decision support system; rule-based system; software architecture; Flask; software project planning; recommendation engine*

I. INTRODUCTION

Software development has grown rapidly in scale and complexity over the last two decades. A project that would once have been implemented with a single server-side language and a relational database can now be built from dozens of competing frontend frameworks, backend runtimes, databases, hosting providers and auxiliary tools. This abundance of choice is generally beneficial, since it allows a technology stack to be tailored to the specific needs of a project; however, it also introduces a non-trivial decision problem for anyone who must choose among these options at the start of a project, when requirements are still being clarified and the consequences of a poor choice are not yet visible.

The importance of selecting a suitable technology stack is well recognised in software engineering practice. The technologies chosen at the outset of a project influence its development speed, its long-term maintainability, its ability to scale with increasing users or data, its security posture, and the overall cost and duration of development. An unsuitable choice — for example, adopting a heavyweight microservices architecture for a two-person student project, or a lightweight monolithic design for a system that is expected to serve a large user base — can result in wasted effort, schedule overruns or a system that fails to meet its non-functional requirements.

Students, early-career developers and small teams are particularly exposed to this problem. Unlike experienced architects, they typically have limited first-hand exposure to multiple technology ecosystems and often lack a structured method for relating project characteristics, such as domain, team size and budget, to an appropriate set of technologies. In academic settings in particular, students frequently select a technology stack based on what was taught in a course, what a tutorial demonstrates, or what is currently popular, rather than through a deliberate evaluation of project requirements. This tends to produce

inconsistent outcomes and does not help students to internalise the reasoning that connects project characteristics to technology and architectural choices.

Viewed formally, technology-stack selection is a multicriteria decision-making problem: given a set of project attributes, a decision maker must choose values along several interdependent dimensions (frontend, backend, database, hosting, architecture) while also estimating derived quantities such as complexity, scalability, development time and cost. Because the criteria interact — for example, team size affects both the recommended architecture and the feasible project roles, while budget affects both the choice of tooling and the achievable scalability — an ad hoc or purely intuitive approach is unlikely to be systematic or repeatable. This motivates the use of an explicit, rule-based decision procedure that can be inspected, justified and modified, rather than a purely subjective judgement.

This paper presents TechManager, an intelligent decision support system that addresses this problem for a defined set of common project domains. TechManager is implemented as a Flask-based web application in which a user supplies a project name, a project type, a team size and a budget, and the system responds with a set of technology recommendations and project-planning outputs derived from a rule-based recommendation engine. The system is intended primarily to support students and small teams during the planning stage of a software project by making the reasoning behind a technology recommendation explicit and by recording a history of past recommendations for later reference. The remainder of this paper describes the problem addressed, related work, the design and implementation of TechManager, its rule-based methodology, a functional evaluation using representative test cases, and the advantages, limitations and future scope of the system.

II. PROBLEM STATEMENT

In the absence of a structured tool, students and small development teams typically select a technology stack through informal means: copying the stack used in a similar tutorial, choosing tools they are already familiar with, or following recommendations found in online forums without relating them to their specific project context. This informal process does not explicitly account for factors that materially affect the suitability of a technology choice, namely the domain of the project (for example, a banking application has different requirements from a portfolio website), the number of people available to build and maintain the system, and the budget available for tools, hosting and third-party services.

The consequence of this gap is threefold. First, the chosen architecture may not match the size of the team, leading either to unnecessary complexity for a small team or to an architecture that cannot be maintained as the team or user base grows. Second, estimates of development time and cost are often absent or based on guesswork, which makes project planning difficult, particularly in an academic project setting where time is bounded by a semester schedule. Third, decisions are rarely explainable: a student following a tutorial cannot easily justify why a particular database or hosting platform was chosen for their specific project.

The problem addressed in this work can be stated as follows: given a project described by its type T (drawn from a fixed set of supported domains), its team size S, and its budget category B, determine a consistent and explainable technology configuration — comprising frontend, backend, database, hosting and architecture — together with supporting project-planning information (complexity classification, scalability indication, estimated development time and cost, team roles, a development roadmap, security recommendations and alternative technologies) that is appropriate to (T, S, B). The solution should be deterministic and explainable, rather than a black-box prediction, so that its output can be understood, taught and adjusted by the people who use it.

III. OBJECTIVES

The objectives of this work are as follows:

- To design a rule-based recommendation engine that maps a project type, team size and budget to a consistent technology configuration (frontend, backend, database, hosting) and software architecture.
- To generate supporting project-planning outputs — complexity classification, scalability score, estimated development time and cost, additional development tools, suggested team roles, a development roadmap, security recommendations and alternative technologies — alongside the core technology recommendation.
- To implement the system as an accessible, responsive web application using a lightweight and freely available technology stack (Flask, HTML, CSS, JavaScript and SQLite), deployed on a cloud hosting platform.
- To persist every generated recommendation for later review, including a history view with search and filter capability and the ability to delete a stored recommendation.

- To keep the recommendation logic transparent and explainable, so that the reasoning behind a suggestion can be inspected and the underlying rules can be modified or extended without redesigning the system.
- To evaluate the system functionally, using representative test cases spanning the supported project domains, team sizes and budget levels, and to document the system's behaviour, advantages and limitations honestly.

IV. LITERATURE REVIEW

Rule-based expert systems have long been used to encode human decision-making knowledge as a collection of condition–action rules that an inference engine evaluates against a set of facts. Jackson's foundational text on expert systems describes how such systems represent domain knowledge explicitly, which allows their reasoning to be traced and explained, in contrast to statistical or learned models whose internal decision boundaries are opaque [1]. This property of explainability is central to the design of TechManager, which similarly encodes technology-selection knowledge as a fixed set of rules rather than learning a mapping from data.

Closer to the specific problem of technology selection, İmamoğlu and Çetinkaya proposed a rule-based decision support system, PLExpert, that guides software engineers in selecting a suitable programming language before the coding stage of a project — the system encodes expert knowledge as roughly thirty conditional rules over a knowledge base of programming-language attributes and produces a language recommendation with an associated justification [2]. Their motivation, that existing project-management tools focus on planning and tracking but provide little support for early technical decisions, closely mirrors the motivation for TechManager, although PLExpert addresses only programming-language selection, whereas TechManager additionally recommends a complete stack (frontend, backend, database, hosting), an architecture and a project plan.

Farshidi et al. present a decision support system for software technology selection that models technology selection as a multi-criteria decision problem and supports users in choosing among competing technology alternatives for a

given software-production context, such as database management systems [3]. Their work emphasises a criteriaand-alternatives structure that can be reused across different technology categories. Jedlitschka, Pfahl and Bomarius similarly propose a framework for experience-based decision support for software engineering technology selection, arguing that decisions about which technologies and methods to adopt should be informed by structured, reusable experience rather than ad hoc judgement [4]. TechManager adopts a comparable philosophy of encoding accumulated, structured knowledge into reusable rules, but targets a substantially narrower and more concrete decision — stack and architecture selection for a small set of common application domains — which allows the resulting rule set to remain simple enough for a lightweight, student-built tool.

Software architecture decision-making has also been studied as a distinct problem from technology selection. Anvaari's doctoral work proposes a rule-based framework for enhancing architectural decision guidance, arguing that architectural decision-making benefits from combining literature-derived and domain-specific rules with the architect's own judgement, rather than leaving the decision entirely to individual experience [5]. TechManager applies an analogous, simplified rule — mapping team size to one of three architectural styles (simple monolithic, modular monolithic, or microservices) — which reflects the intuition documented in this line of work that team size and coordination overhead are primary drivers of architectural suitability, particularly for small teams.

Multi-criteria decision-making (MCDM) techniques, including the analytic hierarchy process and the analytic network process, have also been applied to related softwareselection problems. Akbar et al. propose an MCDM-based framework for selecting software project management tools, in which criteria are weighted and alternatives are ranked using an analytic network process [6]. Such MCDM approaches provide a rigorous way of combining multiple weighted criteria but generally require the user to supply or validate criteria weights and alternative scores, which increases the burden on a non-expert user such as a student.

TechManager instead embeds fixed, predefined rules chosen by its developers, which trades some of the flexibility of a general MCDM framework for simplicity and ease of use for its target audience.

Finally, software project estimation techniques inform the project-planning outputs generated by TechManager. The Constructive Cost Model (COCOMO), introduced by Boehm, remains one of the most widely referenced algorithmic models for estimating software development effort, time and team size from measures of project size and a set of cost-driver attributes [7]. While TechManager does not implement COCOMO directly — its inputs

(project type, team size, budget) do not include a size measure such as lines of code — the same underlying idea, that development time and effort can be estimated from a small number of project attributes through an explicit formula, underlies the time and cost estimation rules used in TechManager.

In summary, existing work establishes that rule-based and criteria-based decision support can assist technology and architecture selection, and that structured effort-estimation models can support project planning. Existing systems, however, tend either to address a single decision (such as programming-language selection alone) in isolation, or to require a level of criteria weighting and alternative evaluation that is better suited to experienced practitioners than to undergraduate student teams. TechManager is distinguished from this body of work by combining a complete technologystack recommendation, an architecture recommendation and a set of project-planning outputs (complexity, scalability, time, cost, roles, roadmap, security, alternatives) within a single lightweight, rule-based tool designed specifically around four simple inputs that a student can supply without prior domain expertise.

V. PROPOSED SYSTEM

TechManager is proposed as a rule-based decision support system that converts a small set of project-level inputs into a

technology recommendation and an accompanying project plan. The system follows a linear conceptual pipeline: Input → Rule Processing → Recommendation → Project Planning Output, as summarised in Fig. 1.

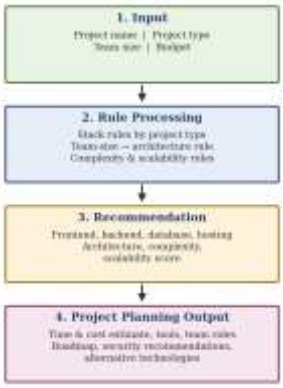


Fig. 1. Conceptual processing pipeline of TechManager, from user input through rule-based processing to recommendation and project-planning output.

In the Input stage, the user supplies a project name (used only for identification and record-keeping), a project type selected from the six supported domains, a team size, and a budget category. In the Rule Processing stage, the recommendation engine evaluates these inputs against predefined rule tables that associate each project type with a base technology profile, each team-size range with an architectural style, and combinations of project type, team size and budget with a complexity classification and a scalability indication. In the Recommendation stage, the engine assembles the core outputs: recommended frontend, backend, database and hosting technologies, the selected software architecture, the complexity category and the scalability score. In the Project Planning Output stage, the engine derives secondary outputs that support planning rather than pure technology choice: an estimated development time, an estimated project cost, a list of additional development tools, suggested team roles, a development roadmap, security recommendations, and a short list of alternative technologies for the same project type. All outputs produced in a single request are stored together as one recommendation record in the SQLite database, allowing the user to revisit, search or delete it later through the web interface.

A key design decision is that every stage of this pipeline is deterministic and rule-driven: for a fixed input (project type, team size, budget), the system always produces the same recommendation. This is a deliberate choice, made so that a

user, or an instructor evaluating the tool, can trace exactly which rule produced which part of the output, rather than having to interpret a probabilistic or learned prediction.

VI. SYSTEM ARCHITECTURE

TechManager follows a layered, monolithic web-application architecture in which each layer has a single responsibility and passes control to the layer below it. The architecture is summarised in Fig. 2.



Fig. 2. Layered system architecture of TechManager: User Interface → Flask Application → Input Validation → Rule-Based Recommendation Engine → Recommendation Results → SQLite Database.

A. User Interface Layer

The user interface is a responsive set of HTML pages, styled with CSS and enhanced with JavaScript for interactive behaviour such as form handling and client-side feedback. This layer exposes the input form for generating a recommendation, a recommendation-details page, a history page with search and filter controls, and an about page. It is intentionally kept independent of the recommendation logic, communicating with the server only through standard HTTP requests.

B. Flask Application Layer

The Flask application layer defines the web routes that handle incoming requests: generating a new recommendation, viewing recommendation details, listing and filtering history, deleting a stored recommendation, serving the about page, and responding to a health-check endpoint used to confirm that the deployed service is running. This layer is responsible for coordinating the remaining layers and for rendering the appropriate HTML templates with the data they require.

C. Input Validation Layer

Before a request reaches the recommendation engine, the submitted project name, project type, team size and budget are validated to ensure that the project type is one of the six supported domains, the team size is a positive integer, and the budget corresponds to one of the recognised budget categories. This layer prevents malformed or out-of-range inputs from reaching the rule engine and allows the application to return a clear error to the user when validation fails.

D. Rule-Based Recommendation Engine

This is the core decision-making layer of the system. It holds the predefined rule tables that associate project types with base technology profiles, team-size ranges with architectural styles, and combinations of inputs with complexity and scalability classifications, and it computes the derived project-planning outputs described in Section V. Because the rules are encoded directly in this layer rather than learned from data, the layer can be modified by editing or extending the rule tables without retraining any model.

E. Recommendation Results Layer

The results produced by the rule engine are assembled into a single recommendation object containing all fourteen output fields described earlier (frontend, backend, database, hosting, complexity, architecture, scalability, development time, estimated cost, additional tools, team roles, roadmap, security recommendations and alternative technologies). This object is passed back to the Flask application layer for rendering and is also handed to the persistence layer for storage.

F. SQLite Database Layer

The final layer persists each recommendation in a SQLite database table named “recommendations”, together with the project name, project type, team size, budget and a creation timestamp. This layer supports the history, search/filter and delete features of the application by allowing past recommendations to be queried and removed.

VII. METHODOLOGY

This section describes, at a formal level, how the ruleprocessing mechanism of TechManager combines its three inputs — project type T, team size S and budget B — to produce its outputs. Because the underlying rule tables are domain-specific lookup structures rather than a single closedform equation, the formulation below should be read as a general description of the rule logic implemented in the system, rather than as the literal source code.

A. Effect of Project Type on Technology Selection

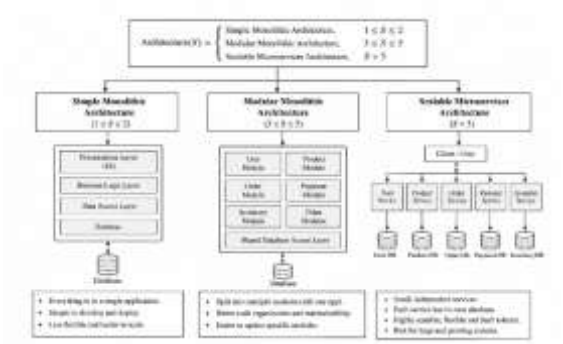
Each supported project type $T \in$ {Portfolio, Education, Hospital, Banking, E-Commerce, E-Sports} is associated, through a static rule table, with a base technology profile

Stack(T) = { Frontend(T), Backend(T), Database(T), Hosting(T) }

This table encodes domain knowledge about which technologies are typically appropriate for a given class of application — for example, a domain with strict dataintegrity and auditing requirements, such as banking, is associated with a base profile that favours a relational database and additional security-oriented tooling, whereas a content-oriented domain, such as a portfolio site, is associated with a lighter-weight profile. The same lookup table also supplies the domain-specific security recommendations, additional development tools and alternative technology suggestions returned to the user.

B. Effect of Team Size on Architecture and Roles

Team size S determines the recommended software architecture through the following piecewise rule, which is applied identically regardless of project type:



This rule reflects the observation, also noted in the architectural decision-making literature reviewed in Section IV, that coordination overhead and the ability to operate independent services both grow with team size: a one- or twoperson team benefits from the reduced operational overhead of a single deployable unit, a small team of three to five can practically manage a modularised monolith, and a larger team can divide responsibility across independently deployable services. Team size also determines the suggested team roles (for example, a very small team is advised to combine frontend and backend responsibilities, while a larger team is advised to separate frontend, backend, database and qualityassurance roles), since the number of distinct roles that can be meaningfully staffed is bounded by S.

C. Effect of Budget on Tools, Cost and Scalability

Budget B ∈ {Low, Medium, High} affects three outputs. First, it determines which additional development tools and hosting tiers are suggested — a higher budget makes managed or paid services eligible for recommendation, while a lower budget restricts suggestions to free or open-source tooling. Second, it acts as a multiplicative adjustment on the base cost and time estimates associated with the project type and complexity, so that a higher-budget project is estimated to afford more tooling and, where relevant, more parallel effort. Third, together with project type and team size, it contributes to the scalability score, since a larger budget generally permits infrastructure choices (e.g., managed hosting, horizontal scaling options) that improve the achievable scalability of the recommended stack.

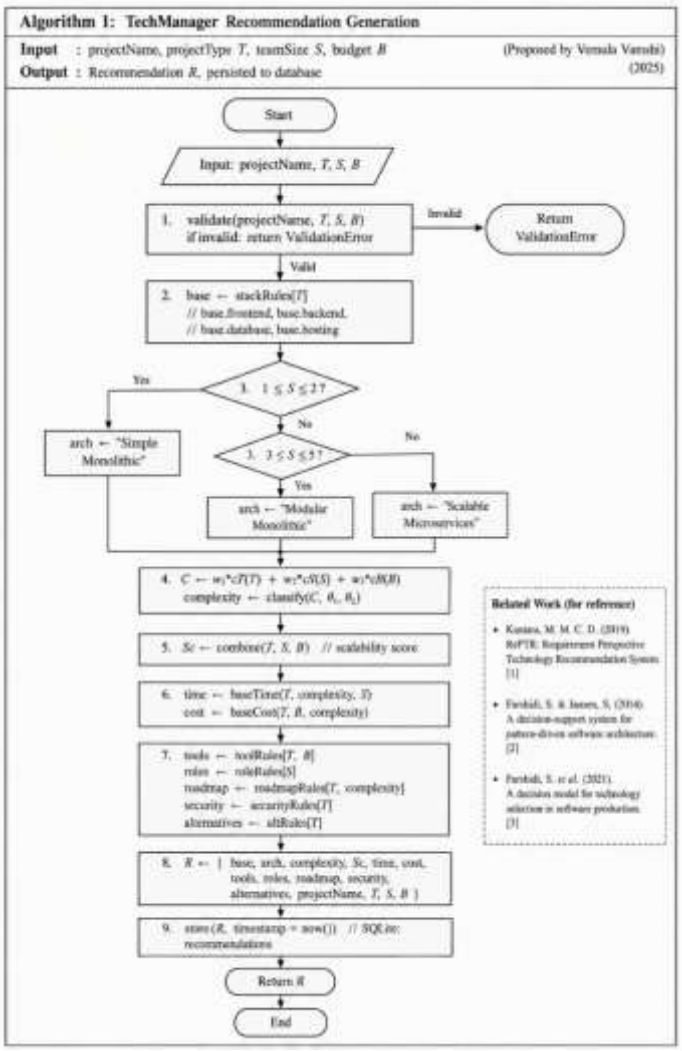
D. Combined Complexity Estimation

The overall project complexity is classified into one of three categories — Beginner, Intermediate or Advanced — based on a combined assessment of project type, team size and budget. This can be expressed conceptually as a weighted combination

$C = w_1 \cdot c_t(T) + w_2 \cdot c_s(S) + w_3 \cdot c_h(B)$ where $c_t(T)$, $c_s(S)$ and $c_h(B)$ are rule-defined weights reflecting how demanding the project type, team size and budget respectively are considered to be, w_1, w_2 and w_3 are fixed importance weights assigned to each factor, and the resulting score C is mapped to Beginner, Intermediate or Advanced through fixed thresholds. A similar weighted combination, evaluated against a different set of rule-defined weights, produces the numeric scalability score reported to the user. This formulation illustrates the general shape of the rule logic; the specific numeric weights and thresholds are internal implementation parameters of the recommendation engine and can be revised as the rule base is refined.

VIII. RULE-BASED RECOMMENDATION ALGORITHM

Algorithm 1 summarise the end-to-end procedure followed by the recommendation engine, consistent with the pipeline and methodology described in Sections V and VII.



Because every function invoked in Algorithm 1 (stackRules, classify, combine, baseTime, baseCost, toolRules, roleRules, roadmapRules, securityRules, altRules) is a deterministic lookup or arithmetic evaluation over predefined tables, the algorithm always produces the

same recommendation for the same input triple (T, S, B). This determinism is what allows the system's behaviour to be verified through the functional test cases presented in Section X, and it is the property that most clearly distinguishes a rule-based engine of this kind from a learned or probabilistic recommender.

IX. IMPLEMENTATION

TechManager is implemented using widely available, free and open-source tools, consistent with its intended use by students and small teams with limited budgets.

A. Backend

The backend is implemented in Python using the Flask microframework. Flask routes correspond to the application's functional features: generating a recommendation from submitted form data, displaying a recommendation-details page, listing and filtering recommendation history, deleting a stored recommendation, serving an about page, and exposing a health-check endpoint that can be used by the hosting platform, or by a developer, to confirm that the deployed application is responsive. The rule-based recommendation engine described in Sections VII and VIII is implemented as plain Python logic within the backend, using conditional statements and lookup structures rather than any statistical or learning-based library.

B. Frontend

The frontend is implemented using HTML for page structure, CSS for styling and responsive layout, and JavaScript for client-side interactivity, such as form validation feedback and dynamic display of recommendation results. The interface is designed to be usable on both desktop and mobile screen sizes.

C. Data Persistence

Recommendations are stored in a SQLite database, in a table named “recommendations”. Each stored record includes the project name, project type, team size, budget, complexity classification, recommended frontend, backend, database and hosting technologies, architecture, scalability score, estimated development time, estimated cost, and the date and time the record was created. SQLite was selected because it requires no separate database server, ships as a single file, and is well suited to a single-instance prototype deployment of this kind.

D. Version Control and Deployment

Source code for TechManager is version-controlled using Git and hosted on GitHub, which supports incremental development and collaboration among the team members. The application is deployed on Render, a cloud platform that builds and runs the Flask application from the GitHub repository and exposes it as a publicly accessible web service. The health-check endpoint mentioned above is used to verify that the Render deployment is live.

No component of the implementation described above uses machine learning, deep learning, generative artificial intelligence or neural networks; all recommendation logic is rule-based, as described in Sections VII and VIII. Possible learning-based extensions are discussed only as future work, in Section XIV.

X. EXPERIMENTAL SETUP AND TEST CASES

Because the recommendation engine is deterministic and rule-based rather than statistical, the system was evaluated functionally: a set of representative test cases was constructed to span the six supported project types together with varying team sizes and budget levels, and the outputs produced by the engine for each test case were checked for internal consistency with the rules described in Sections VII and VIII (for example, that the reported architecture always matches the team-size rule, and that complexity and scalability move in the expected direction as team size and budget increase). Table I lists the six test cases used, each identified by a project type, a team size and a budget level.

ID	Project Type	Team Size	Budget
T1	Portfolio	1	Low
T2	Education	4	Medium
T3	Hospital	5	High
T4	Banking	8	High
T5	E-Commerce	3	Medium
T6	E-Sports	6	High

TABLE I. Input Combinations Used for Functional Test Cases

For each test case, the architecture output is fully determined by the team-size rule in Section VII-B and can therefore be verified exactly. The complexity and scalability outputs are expected, by design of the rule engine, to increase as project type demands grow (e.g., banking and hospital domains carry stricter data-handling expectations than a portfolio site) and as team size and budget increase. Table II reports the architecture obtained for each test case together with the qualitative complexity and scalability trend expected from the rule design; Table

III reports the qualitative developmenttime and cost trend for the same test cases.

ID	Architecture (from S)	Complexity (expected)	Scalability (expected)
T1	Simple Monolithic	Beginner	Low
T2	Modular Monolithic	Intermediate	Medium
T3	Modular Monolithic	Advanced	High
ID	**Architecture (from S)**	**Complexity (expected)**	**Scalability (expected)**
T4	Scalable Microservices	Advanced	High
T5	Modular Monolithic	Intermediate	Medium
T6	Scalable Microservices	Intermediate–Advanced	Medium–High

TABLE II. Architecture, Complexity and Scalability Obtained for Each Test Case

ID	Development Time (trend)	Estimated Cost (trend)
T1	Shortest of six cases	Lowest of six cases
T2	Moderate	Moderate
T3	Longer (high complexity)	Higher (high budget)
T4	Longest (largest team)	Highest of six cases
T5	Moderate	Moderate
T6	Moderate–long	Higher (high budget)

TABLE III. Development Time and Cost Trends Obtained for Each Test Case

Each test case was executed against the deployed application, and the returned architecture value was confirmed to match the deterministic team-size rule in every case; the complexity, scalability, time and cost outputs were confirmed to move in the qualitative directions summarised in Tables II and III as project demands, team size and budget increased. As the recommendation engine does not draw on any external, realworld dataset of completed projects, no claim is made, and none should be inferred, about how closely these estimated values correspond to the actual time or cost of a real project of the same description; the evaluation reported here is functional and test-case based, verifying that the engine behaves consistently with its own encoded rules rather than benchmarking it against real-world outcomes.

XI. RESULTS AND DISCUSSION

The functional evaluation in Section X shows that, for every test case, the TechManager recommendation engine produced an architecture, complexity and scalability classification consistent with the rules described in Sections VII and VIII, and that repeated submission of the same input triple (T, S, B) produced identical output on every run. This determinism and consistency is an expected and desirable property of a rule-based system: because there is no random initialisation, sampling or model training involved, the same input always follows the same path through the rule tables and arithmetic combinations described in Algorithm 1.

A further observation is that the reasoning behind each recommendation is explainable by construction. For any generated recommendation, it is possible to state precisely which rule or rule combination produced each output field — for instance, that the architecture recommendation for test case T4 follows directly from the team-size rule for S = 8, or that the higher complexity and scalability values for the banking and hospital domains follow from the higher domain weights $c_t(T)$ assigned to those project types. This traceability is a direct consequence of the rule-based design and stands in contrast to a machine-learning-based recommender, whose output for a given input generally cannot be attributed to a single, human-readable cause without additional explainability tooling.

From a project-planning perspective, the results suggest that TechManager is useful primarily as a starting point for discussion and planning rather than as a final authority. A student team can use the generated architecture, role suggestions and roadmap as an initial plan, which they are still expected to review and adapt to the specifics of their project; the system does not attempt to replace this judgement, only to make an informed starting point available quickly and consistently. The discussion above should be read alongside the limitations noted in Section XIII, in particular the fact that the evaluation performed in this paper is functional and does not constitute a validated measurement of recommendation quality against real-world project outcomes.

XII. ADVANTAGES

- Explainability: because every output is produced by an inspectable rule or a simple weighted combination of rule-defined factors, the reasoning behind a recommendation can always be traced and explained to the user.
- Fast recommendations: rule lookups and arithmetic combinations execute almost instantly, so the user receives a complete recommendation and project plan within a single request, without the latency associated with training or invoking a large predictive model.
- Simple implementation: the recommendation engine is implemented with ordinary conditional logic and lookup tables in Python, requiring no specialised machine-learning infrastructure, training data or modelserving pipeline.
- Easy rule modification: because domain knowledge is stored as explicit rule tables, the development team can add a new project type, adjust a threshold, or revise a technology suggestion by editing the relevant table, without retraining or redesigning the system.
- Multi-factor decision making: the engine jointly considers project type, team size and budget when producing its architecture, complexity, scalability, time and cost outputs, rather than basing the recommendation on a single factor in isolation.
- Project planning support: beyond a bare technology list, the system also produces team roles, a development roadmap and security recommendations, which are

directly useful to a student team preparing a project plan.

XIII. LIMITATIONS

- The quality of every recommendation is entirely dependent on the predefined rules encoded by the development team; if a rule reflects an outdated or incomplete understanding of a domain, the resulting recommendation will inherit that limitation.
- The system does not learn from historical data or from feedback on past recommendations; it cannot improve its rules automatically as it is used, unlike a data-driven recommender that can be retrained.
- Because the underlying technology landscape changes over time, the rule tables require periodic manual review and updating to remain relevant; a rule-based system carries an ongoing maintenance cost that a selfupdating system would not.
- The system currently supports only six project domains (portfolio, education, hospital, banking, e-commerce and e-sports); a project that does not fit clearly into one of these categories cannot be meaningfully assessed by the current rule set.
- SQLite is adequate for a single-instance prototype but is not designed for concurrent multi-user, multi-writer production workloads; a larger-scale deployment would require a more robust, server-based database.
- The recommendations produced by TechManager are intended as decision support rather than a guarantee of the optimal technology choice; the final decision, and responsibility for adapting the suggestion to the specifics of a given project, remains with the user.

XIV. FUTURE SCOPE

Several extensions could improve TechManager beyond its current rule-based scope, and are identified here explicitly as future work rather than as part of the current implementation.

- Machine-learning-based recommendation: incorporating a learned model trained on outcomes from completed projects could complement, or eventually extend, the current rule-based engine, provided that sufficient labelled project data becomes available.
- A larger technology knowledge base covering a wider range of frontend, backend, database and hosting options, rather than the currently predefined per-domain profiles.
- Support for additional project domains beyond the current six, to widen the range of projects the system can meaningfully assess.
- Integration of real-time technology information, such as current library popularity, maintenance status or

pricing, rather than relying solely on static, manually curated rules.

- Developer skill matching, so that recommendations can be adjusted to the specific skills already possessed by the members of a given team.
- Cloud-cost comparison across multiple hosting providers, to give a more concrete and current cost estimate than the current rule-based approximation.

- Performance and security benchmarking of recommended stacks, to supplement the current rulebased security recommendations with measured evidence.
- Migration from SQLite to PostgreSQL, or a comparable server-based database, to support multi-user, production-scale deployment.
- Personalised recommendations that take into account a user's or team's history of past projects and preferences, rather than treating every request independently.

XV. CONCLUSION

This paper presented TechManager, a rule-based decision support system that helps students and small development teams select a technology stack and plan a software project from four simple inputs: project name, project type, team size and budget. The system's recommendation engine encodes domain knowledge for six project types as explicit rules governing technology selection, architecture, complexity and scalability, and combines this with straightforward estimation rules for development time, cost, team roles, a development roadmap, security recommendations and alternative technologies. The system is implemented as a

Flask web application backed by a SQLite database and deployed on Render, and its behaviour was evaluated functionally against a representative set of test cases spanning all supported project domains, team sizes and budget levels. The central contribution of this work is not a claim that TechManager identifies the universally optimal technology stack for a given project — no rule-based system operating on three coarse inputs could credibly make such a claim — but rather a demonstration that a transparent, explainable and easily maintainable rule-based tool can provide a consistent, well-reasoned starting point for technology-selection and project-planning decisions that students and small teams currently make informally. The limitations discussed in Section XIII, particularly the system's dependence on manually maintained rules and its restriction to six project domains, define the boundaries within which this contribution should be understood, and motivate the future extensions outlined in Section XIV.

XVI. REFERENCES

[1] P. Jackson, Introduction to Expert Systems, 3rd ed. Harlow, U.K.: Addison-Wesley, 1999.

[2] M. Y. İmamoğlu and D. Çetinkaya, “A rule based decision support system for programming language selection,” in Proc. 2017 2nd Int. Conf. on Knowledge Engineering and Applications (ICKEA), London, U.K., 2017, doi: 10.1109/ICKEA.2017.8169904.

[3] S. Farshidi, S. Jansen, R. de Jong, and S. Brinkkemper, “A decision support system for software technology selection,” Journal of Decision Systems, vol. 27, sup. 1, pp. 98–110, 2018, doi: 10.1080/12460125.2018.1464821.

[4] A. Jedlitschka, D. Pfahl, and F. Bomarius, “A framework for experience-based decision support for software engineering technology selection,” in Proc. 16th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE 2004), Banff, Canada, 2004, pp. 342–345.

[5] M. Anvaari, “A rule-based framework for enhancing architectural decision guidance,” Ph.D. dissertation, Dept. of Computer Science, Norwegian Univ. of Science and Technology (NTNU), Trondheim, Norway, 2016.

[6] S. Akbar, R. Ullah, R. Khan, I. Asghar, M. Zubair, and Z. Zheng, “A multi-criteria decision-making framework for software project management tool selection,” in Proc. 2023 9th Int. Conf. on Computer Technology Applications (ICCTA 2023), Vienna, Austria, ACM, 2023, doi: 10.1145/3605423.3605454.

[7] B. W. Boehm, Software Engineering Economics. Englewood Cliffs, NJ, USA: Prentice-Hall, 1981.

[8] Pallets Projects, “Flask Documentation,” flask.palletsprojects.com, accessed Sept. 2026.

[9] SQLite Consortium, “SQLite Documentation,” sqlite.org, accessed Sept. 2026.