University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

ArchHypo Tooling

Enabling Practical Hypothesis Engineering for Software Architecture

Pedro Henrique Mariano Corrêa

Final Essay

mac 499 — Capstone Project

Supervisor: José Gonçalves Lima Neto
Co-supervisor: Prof. Dr. Paulo Roberto Miranda Meirelles

Acknowledgements

I would like to express my deep appreciation to Professor Paulo Meirelles and Jose Neto for their invaluable support and advising throughout the development of this capstone project. Their guidance was essential to realizing this work.

I am also grateful to the Eduardo Guerra group, whose research provided the main foundation for this project. Thank you for the generosity of your time and the insightful conversations that helped shape our approach.

Finally, I thank my family for their encouragement. A special thank you goes to my mother for the enduring life lessons that have brought me to this moment. To the friends I have made along this university journey, thank you for being a part of my story.

Abstract

Pedro Henrique Mariano Corrêa. **ArchHypo Tooling: *Enabling Practical Hypothesis Engineering for Software Architecture***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo.

This capstone project addresses the operational gap in modern software architecture, where rapid evolution often lacks methodological support for decentralized decision-making. While industry trends favor autonomous teams, architectural choices often remain dependent on the tacit knowledge of experienced practitioners. To resolve this, the work builds upon the ArchHypo framework, which treats software architecture as a set of testable hypotheses to manage impact and uncertainty.

Despite its theoretical benefits, ArchHypo's manual application imposes a high cognitive load and significant adoption barriers for development teams. This project introduces HypoStage, an open-source plugin for the Backstage Internal Developer Portal (IDP), designed to operationalize hypothesis engineering. HypoStage integrates directly into the developer's workflow, providing structured digital tools for uncertainty assessment, quality attribute tracking, and technical planning. The practical utility of the tool is demonstrated through a case study involving a web testing start-up, illustrating how it supports the full lifecycle of architectural elicitation, assessment, and validation. Ultimately, HypoStage provides the technological scaffolding necessary to bridge the gap between architectural theory and the reality of continuous delivery.

Keywords: Software Architecture. Hypothesis Engineering. ArchHypo. Backstage.

List of Abbreviations

SA	Software Architecture
ArchHypo	Architecture Hypothesis
IDP	Internal Developer Portal
NFR	Non-Functional Requirements

Contents

Chapter 1

Introduction

Modern software engineering faces a significant conflict between the demand for rapid feature delivery and the necessity of maintaining a stable, sustainable Software Architecture (SA). While agile methodologies have successfully introduced practices for functional evolution, they provide sparse guidance for architectural evolution. This lack of recognized methodological support often forces teams into a precarious choice: either performing heavy upfront designs that contradict agile principles or allowing the system to degrade into a "Big Ball of Mud" where architectural integrity is sacrificed for speed.

This methodological deficiency is compounded by a critical lack of operational tooling to support architectural decision-making. As identified in the systematic mapping by Souza *et al.*, 2019, existing derivation methods rely heavily on the tacit knowledge of experienced architects, offering little support for less experienced practitioners or decentralized teams. Furthermore, research by Silva, Melegati, Silveira, *et al.*, 2025 emphasizes that even when robust frameworks like ArchHypo are introduced to manage architectural uncertainty, their manual application remains difficult and cognitively demanding. Consequently, a "tooling void" exists where architectural management remains disconnected from the daily engineering workflow, preventing teams from effectively bridging the gap between high-level theory and the operational reality of continuous delivery.

1.1 Software Architecture and the Challenge of Rapid Evolution

In the modern landscape of software engineering, SA has evolved from a static blueprint into a continuous, dynamic process that is critical for business survival. As systems grow in complexity, SA serves as the structural foundation that ensures long-term sustainability, acting as a strategic asset that dictates an organization's agility in fluctuating markets.

Traditionally, architecture was defined by Perry and Wolf, 1992 tripartite model comprising elements (processing, data, and connections), form (relationships and weights), and rationale (justification). However, the modern pressure to deliver quickly often obfuscates the rationale, causing the form to degrade into what is known as a "Big Ball of Mud", as

described by Foote and Yoder, 2003.

The core tension today lies in balancing velocity with stability. While architecture was once front-loaded, maintaining integrity while enabling rapid feature delivery is now crucial, particularly in fast-paced environments like startups where requirements emerge frequently. Practitioners must balance speed with quality to ensure adaptable architectures; failing to do so results in technical debt that paralyzes future development.

1.2 The Operational Gap: Agility Without Support

While the software industry has embraced modern trends such as DevOps and Team Topologies to empower autonomous teams and accelerate value flow, a significant gap remains between these agile aspirations and the methodological support available for architectural decision-making.

The industry increasingly demands that architecture evolve continuously alongside code. However, existing methods for deriving and managing architecture often lag behind, remaining abstract, manual, or disconnected from the daily development workflow. There is a distinct lack of tooling that allows teams to systematically manage architectural evolution without slowing down the development pipeline. Without integrated support, teams struggle to apply architectural rigor in real-time, often resorting to ad-hoc decisions that bypass long-term structural considerations.

This deficiency creates a barrier to true agility. Organizations attempt to decentralize ownership to handle fast-flowing requirements, but they lack the concrete instruments to guide these decisions effectively. Consequently, the evolution of software architecture becomes a friction point rather than an enabler, highlighting the urgent need for frameworks and tools that bridge the gap between high-level architectural theory and the operational reality of continuous delivery.

1.3 The Philosophical Solution: ArchHypo

To address this operational challenge, this work builds upon the principles of ArchHypo, a technique that shifts the view of architecture from a set of facts to a set of experiments. ArchHypo employs hypotheses engineering to manage the uncertainties inherent in software architecture.

By explicitly documenting uncertainties, ArchHypo allows for the strategic postponement of decisions. This aligns with the "Continuous Architecture" principle of delaying design decisions until the "Responsible Moment"—the point where the cost of deciding is outweighed by the cost of waiting, and sufficient information is available to minimize risk. Rather than making premature commitments that lead to brittle systems, teams formulate technical plans based on hypothesis assessment to mitigate impact and reduce uncertainty. This evidence-based process allows decision-making to be safely distributed among team members.

1.4 The Practical Contribution: From Theory to Tooling

While ArchHypo offers a robust theoretical framework, its practical application faces significant friction due to the high cognitive load required to shift from a feature-centric to a hypothesis-centric mindset. Observational studies indicate that without guidance, teams find the technique hard to learn, particularly regarding mapping risks and defining action plans.

There is a strong need for tool support to manage these hypotheses and execute action plans, moving architectural management out of abstract documentation and into the engineering lifecycle. To address this, the primary practical contribution of this capstone is the design and development of HypoStage. As an ArchHypo plugin for Backstage, HypoStage integrates directly into the developer's existing workflow within an Internal Developer Portal (IDP).

HypoStage operationalizes ArchHypo by providing digital structures for uncertainty assessment, quality attribute tracking, and technical planning. By embedding these capabilities into a tool, this work aims to bridge the gap between theoretical agility and the practical reality of fast-paced development, providing the necessary infrastructure to support continuous architectural evolution.

Chapter 2

Theoretical and Architectural Basis

This chapter establishes the theoretical foundations necessary for the development of the proposed solution. It begins by contextualizing Software Architecture within dynamic and agile environments, identifying the friction between rapid delivery and architectural integrity. Subsequently, it delineates the central problem motivating this work: the dissonance between distributed team structures and centralized decision-making methods. To address this gap, the chapter introduces the concept of Internal Developer Portals (IDPs) as essential infrastructure for team autonomy. Finally, it details the principles of Hypothesis Engineering and the ArchHypo framework, justifying the necessity for specific tooling to operationalize these practices.

2.1 The Role of Software Architecture in Agile and High-Flow Environments

Software Architecture (SA) is a fundamental concept in software development, forming the structural foundation of systems and ensuring their long-term sustainability. While early definitions characterized SA as a composition of elements, form, and rationale, modern definitions have evolved to incorporate deeper insights into the dynamics of system evolution. Contemporary definitions characterize SA as a high-level system structure encompassing essential characteristics, design principles, and the decisions to guide system adaptability and maintainability.

This expanded perspective highlights that SA is not merely a static blueprint but a dynamic framework that guides decisions throughout the system life cycle and aligns technical and business needs. However, the practical application of architectural design is fraught with complexity, particularly regarding the timing and certainty of decisions.

2.1.1 The Challenge of Architecture in Agile Contexts

Agile methodologies have successfully introduced practices like Test-Driven Development (TDD) and refactoring, yet they often lack well-recognized and agreed-upon

approaches for architectural design. In many agile contexts, the most recognized approach is to simply let the architecture emerge and refine it over the life of the project. While this approach aims to reduce upfront design, it often leads to a lack of established practice, meaning architectural changes remain challenging within agile contexts, often necessitating significant upfront design that contradicts core agile principles.

In rapidly evolving business environments, such as software startups, new requirements continually reshape architectural demands. In these dynamic scenarios, architecture anti-patterns—such as the "Big Ball of Mud"—may emerge, posing risks to effective architectural evolution and resulting in systems that struggle to maintain proper modularity and scalability.

This anti-pattern often emerges when immediate demands and incremental fixes override deliberate architectural planning. Consequently, engineering teams must balance speed with quality to ensure sustainable, adaptable architectures that meet evolving business demands.

2.1.2 Trends in Evolutionary Architecture

To address the friction between agility and structure, concepts such as *Agile Architecture* and *Continuous Architecture* have emerged. These frameworks emphasize principles such as delaying design decisions and architecting for change. The core philosophy is that decisions can be postponed until they are absolutely necessary, ensuring that architectures are based on facts rather than guesses.

However, determining which decisions to delay or where flexibility is beneficial incurs significant challenges. Time emerges as a critical dimension in architectural design, with iterative planning being essential, particularly for solutions undergoing continuous evolution.

2.1.3 The Necessity of Fast Flow

Recent industry trends emphasize the concept of fast flow, which refers to an organization's ability to continuously and rapidly deliver software changes that align with evolving business needs while preserving the system health and architectural integrity. Achieving fast flow relies on autonomous, empowered teams supported by self-service platforms that minimize operational blockers.

This concept is intrinsically linked to the adoption of socio-technical architecture, which reflects the growing realization that architectural decisions should not be confined to a select few architects but distributed across development teams. The socio-technical approach promotes team autonomy and empowerment, advocating for decision-making processes that include all team members, regardless of their experience levels.

2.2 The Dissonance in Architectural Decision-Making

Despite the industry's aspiration for decentralized, autonomous teams capable of maintaining a fast flow of requirements, a significant theoretical and practical barrier exists. We identify this barrier as the operational gap between intent and method.

2.2.1 The Conflict: Autonomy vs. Tacit Knowledge

A fundamental conflict exists between industry reports advocating for decentralized, team-driven architectural decisions and academic research indicating that existing methods for deriving SA still heavily depend on the tacit knowledge of experienced practitioners.

While the industry moves toward distributed architectural ownership, the current methods and frameworks continue to reinforce centralized decision-making. This creates a self-reinforcing cycle: while organizations may aim for a decentralized approach to support a fast flow of requirements, they remain dependent on expert-driven methods that, by design, limit the involvement of less experienced team members.

2.2.2 Reliance on Tacit Knowledge

A systematic mapping study by Souza *et al.*, 2019 examined methods and practices for deriving architectural models from requirements specifications. A key finding was that existing methods strongly relied on experienced practitioners' tacit knowledge to derive the architectural definitions for a software system. This reliance on the intuition and expertise of architects makes it difficult for teams without such expertise to adopt these methods effectively.

This dependency creates bottlenecks that limit an effective decentralization of decision-making. As a result, the shift towards a more distributed decision-making model is obstructed by frameworks that inherently require centralization, blocking the practical adoption of a socio-technical architecture approach.

2.2.3 Deficiencies in Existing Methods

Beyond the reliance on tacit knowledge, the systematic mapping study highlighted several other gaps in existing architectural derivation methods:

- **Lack of Tool Support:** About a third of analyzed architectural derivation approaches lack tool support. The study noted that only 30.7% of methods provided decision-making support.
- **Inadequate Support for Non-Functional Requirements (NFRs):** Although NFRs such as performance, scalability, and security are essential, many approaches focus primarily on functional requirements, neglecting NFRs in architectural decision-making processes.

- **Limited Empirical Validation:** Many methods have not been sufficiently validated in real-world, dynamic environments. Over half lack explicit evaluation methods.

This points to a clear need for tools to aid architects, suggesting patterns and supporting decision-making to reduce dependency on seasoned experts.

2.3 Internal Developer Portals and Backstage

To bridge the gap between autonomous teams and the complex ecosystem of modern software development, the industry has turned toward *Platform Engineering* and the implementation of Internal Developer Portals (IDPs).

2.3.1 The Role of Developer Portals

An Internal Developer Portal (IDP) serves as the primary interface between developers and the underlying platform infrastructure. Unlike a simple wiki or a project management tool, an IDP is an operational hub that aggregates tools, services, documentation, and data into a unified "pane of glass."

The primary goal of an IDP is to reduce cognitive load. By abstracting the complexity of infrastructure and standardizing workflows, IDPs enable "Golden Paths"—recommended, supported, and automated paths for building and deploying software. This aligns directly with the goal of decentralized architecture: providing teams with the autonomy to build and own their services while ensuring governance and best practices are baked into the tooling.

2.3.2 Backstage: The Open Platform for Building Developer Portals

Backstage is an open-source framework for building Internal Developer Portals (IDPs), originally developed by Spotify and now hosted by the Cloud Native Computing Foundation (CNCF). Due to its extensibility and "catalog-first" philosophy, it has emerged as the de facto standard for developer portals.

Architecture and Core Components

As illustrated in Figure 2.1, Backstage is not a rigid application but rather a modular ecosystem driven by configuration. The architecture is divided into several key interaction layers:

- **The Core Framework:** The central hub of the application which manages essential utilities such as the *Plugin Loader*, *Router*, *Auth Service*, and *Error Handling*. It acts as the bridge between the configuration and the varied plugins.
- **Plugin Ecosystem:** Backstage relies heavily on a split-plugin architecture:
  - **Frontend Plugins:** Handle Views, UI Components, and API Clients to present data to the user.

  - **Backend Plugins:** Manage API Services, Data Processing, and Storage, communicating directly with the core framework.

- **External Integrations:** The Backend Plugins serve as the gateway to *External Systems & APIs*, allowing the portal to aggregate data from cloud providers, CI/CD tools, and other infrastructure.

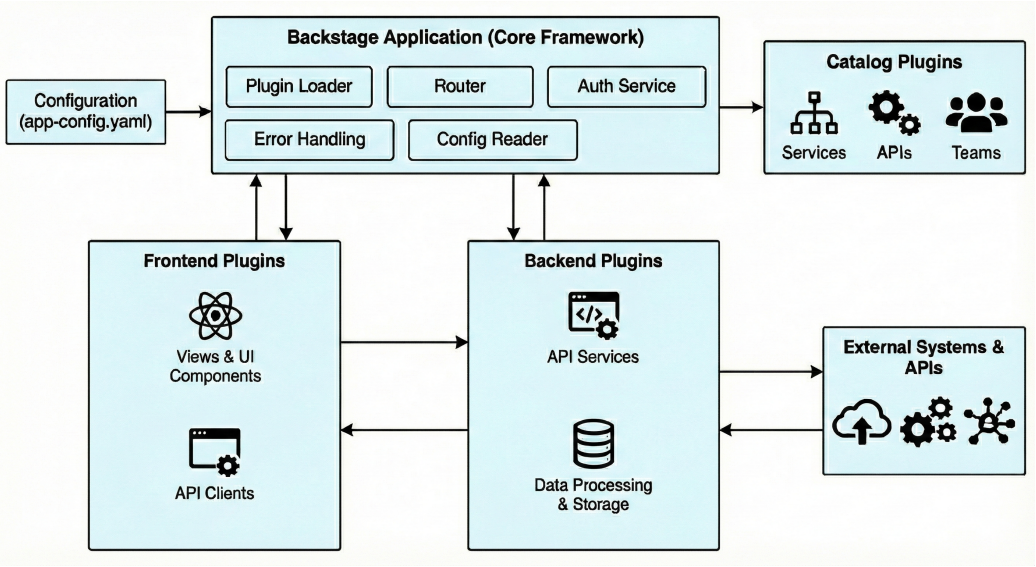

Figure 2.1: *The Backstage organization and platform structure*

Crucially, Backstage is built on a plugin architecture. This allows organizations to extend the portal's functionality by integrating custom tools directly into the developer's daily workflow without forcing developers to switch contexts.

2.4 Hypothesis Engineering and Uncertainty Management

To resolve the decision-making gap and enable decentralized architecture, it is necessary to move away from reliance on tacit intuition toward a more explicit management of the unknowns. This is achieved through **Hypothesis Engineering**.

2.4.1 The Concept of Hypothesis Engineering

Hypothesis Engineering offers a philosophical approach to managing uncertainty by treating assumptions as testable hypotheses. It is defined as a process of continuously validating product assumptions, transforming them into hypotheses, prioritizing and testing them following the scientific method to support or refute them.

In this context, the word "hypothesis" focuses on business assumptions that should be evaluated before start-ups develop their business models. However, this concept is extended here to manage the uncertainty of architectural decisions.

2.4.2 Distinguishing Uncertainty from Risk

A critical theoretical distinction must be made between **risk** and **uncertainty**. Risk involves evaluating the probability and impact of specific events; it represents the likelihood of an event occurring and its potential consequences.

In contrast, **uncertainty** refers to the lack of information necessary for architectural decisions. Uncertainty represents the lack of information to make a decision; in cases where partial information is available, the uncertainty is related to what is still unknown. ArchHypo is based on the premise that uncertainties related to the software architecture are natural in all stages of a software project and that, instead of resisting them, a better approach would be to embrace and manage them.

2.4.3 Characteristics of an Architectural Hypothesis

An architectural hypothesis represents any uncertain statement relevant to the software architecture design. Its primary characteristic is that it must be **falsifiable**—that is, the possibility of proving it false exists.

Unlike a requirement statement, which is usually assumed as true when written, using the word hypothesis makes it clear to the whole team that it represents uncertainty. Hypotheses must be shared with the development team so that everyone is aware of the uncertainties that can affect architectural decisions. This explicit documentation allows the team to systematically test and validate assumptions, gather data, and adjust their approach as needed.

2.5 The ArchHypo Framework

ArchHypo is a technique that uses hypothesis engineering to manage uncertainties related to software architecture and enhance decision-making processes. It provides a structured framework for software architects and developers, enabling them to make uncertainties explicit and manageable.

2.5.1 Sources of Uncertainty

When applying ArchHypo, a team focuses specifically on hypotheses that can affect the software architecture. The most common sources of uncertainty are in the **requirements** and the **solutions**.

- **Requirements Uncertainty:** This occurs when a requirement is based on limited evidence or lacks important details. For instance, uncertainty regarding the number of simultaneous requests a system must handle.
- **Solution Uncertainty:** This relates to the unknown consequences of current architecture components or candidate solutions. For example, uncertainty regarding whether a specific library is compatible with a required protocol.

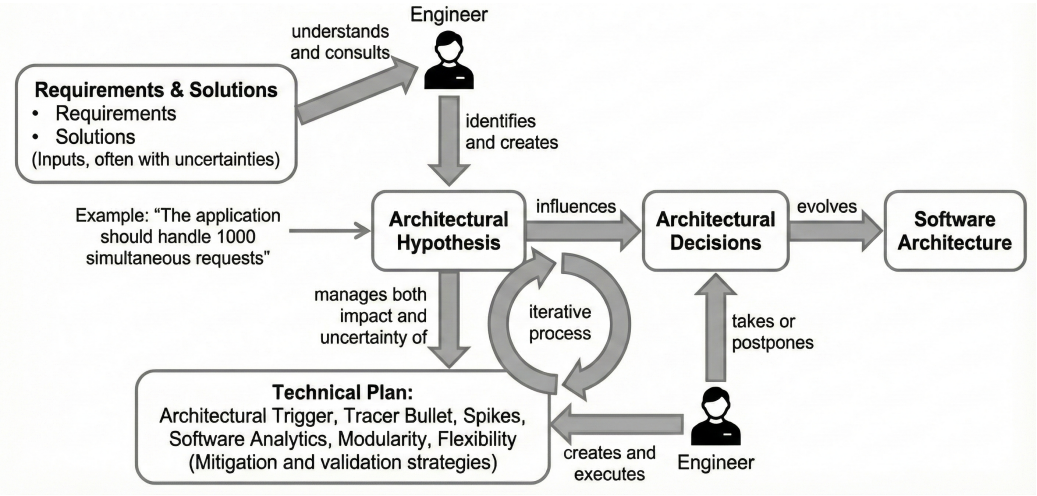

Figure 2.2: *The ArchHypo decision management lifecycle*

2.5.2 Assessment: Uncertainty Level and Impact Level

Once a hypothesis is identified, it is assessed based on two independent dimensions:

- **Uncertainty Level:** This reflects how far the team is from proving that a hypothesis is true or false. The uncertainty would be high if there is a lack of information to estimate probability or when alternatives have a similar chance to happen.
- **Impact Level:** This measures the effort required to transition to a different alternative. It represents the consequences and costs that the uncertainty can cause.

This assessment is typically performed qualitatively by the team, using a scale (e.g., a five-point Likert scale: Very Low to Very High). These assessments aim to provoke reflection about the hypotheses, allowing a qualitative comparison among them and helping in choosing techniques for handling the respective uncertainty.

2.5.3 The Technical Plan

The core operational mechanism of ArchHypo is the formulation of a **Technical Plan** based on each hypothesis' assessment. This plan is derived from the pattern *Plan for Responsible Moments*.

The plan can include actions aiming to:

- Definitely accept or refute the hypothesis.
- **Reduce the uncertainty:** Giving the team more confidence to move forward with a decision.
- **Reduce the impact:** Allowing the team to stay longer with the uncertainty by isolating the affected areas or increasing flexibility.
- Define criteria (triggers) to postpone handling the hypothesis until a specific condition is met.

2.5.4 Supporting Practices and Patterns

The technical plan utilizes a specific pattern language to address uncertainties. Key patterns include:

- **Architectural Spike:** Small technical experiments in which working software is created to prove or disprove the feasibility of a specific hypothesis.
- **Tracer Bullet:** An architecturally significant functionality that helps to exercise and demonstrate an end-to-end path inside the architecture, aiming to evaluate how new technologies could be integrated. This pattern provides a concrete way to design the basic application architecture.
- **Software Analytics:** Identifying relevant metrics to collect and analyze to assess and monitor a given quality attribute of the system.
- **Architectural Trigger:** Defines conditions that trigger architectural investigations which may lead to adding tasks to the backlog. For example, having a given number of simultaneous accesses might fire a trigger related to scalability.
- **Development Guidelines:** Small adjustments in the development process to deal with recurrent uncertainties.
- **Protective Guideline:** Defining programming practices to be followed or avoided to not limit options for an architectural decision being postponed.
- **Bring the Specialist:** Involving individuals with the right skills or knowledge in activities where this expertise can reduce the uncertainty.
- **Plan for Preparation:** Introducing steps to obtain information before activities that recurrently have an associated uncertainty.
- **Quality Checkpoint:** Introducing a verification activity after the development of an artifact to verify if the desired quality is present.

2.6 The Practical Gap and the Need for Tooling

While the theoretical framework of ArchHypo is robust, empirical evidence suggests that its manual application presents significant challenges that hinder widespread adoption.

2.6.1 Adoption Barriers

The empirical study in Silva, Melegati, Silveira, *et al.*, 2025 found that, although the technique offered a structured approach to dividing architectural work, the team identified significant challenges in its adoption due to the learning curve and required process adjustments. In particular, participants reported difficulty in learning the method, especially with respect to mapping risks, specifying hypotheses, and defining action plans.

Team members mentioned difficulty in mapping the scenarios with risk to the hypothesis and definition of the actions to handle the hypothesis. One participant explicitly noted

that "train[ing] the team a little more on how to find, map risk scenarios, and define the necessary actions... would make the team less dependent on the team of architects".

2.6.2 The Integration Necessity

The identified difficulties highlight a strong need for better guidance and, crucially, **tool support to manage hypotheses and execute action plans**. However, standalone tools often suffer from low adoption because they require developers to leave their primary workflow.

This justifies the decision to integrate ArchHypo into an IDP like Backstage. By embedding hypothesis management into the same platform where developers already manage services, documentation, and deployments, we can reduce the cognitive load of adoption. This integration—operationalized in the proposed **HypoStage** plugin—aims to bridge the gap between theoretical agility and the practical reality of fast-paced development.

Chapter 3

The HypoStage Tooling Solution

This chapter presents the practical contribution of this research: the conception, architectural design, and implementation of **HypoStage**. Based on the theoretical foundations established in the previous chapter and the empirical gaps identified in the literature regarding the ArchHypo framework, this chapter details the development of a software tool designed to operationalize decentralized architectural decision-making. The chapter begins by justifying the necessity of a technological intervention to resolve the operational gap between architectural intent and execution. It then proceeds to describe the solution's integration within the Backstage internal developer portal, details the functional requirements for managing the hypothesis lifecycle, and concludes with a technical examination of the software architecture and the open-source strategy adopted for the project.

3.1 Justification for Tooling Support

The theoretical framework of ArchHypo, as discussed in Chapter 2, provides a robust methodology for managing architectural uncertainty through Hypothesis Engineering. However, the transition from theory to practice presents significant hurdles. Empirical research conducted by Silva, Melegati, Silveira, *et al.*, 2025. on the application of ArchHypo has demonstrated clear benefits, such as improved collaboration and a structured approach to architectural work, yet it simultaneously revealed significant adoption challenges.

3.1.1 The Dilemma of Manual Application

The primary barrier to the widespread adoption of Hypothesis Engineering in software architecture lies in the cognitive load and process overhead imposed on development teams. Participants in empirical studies found the technique hard to learn, particularly in mapping risks, specifying hypotheses, and defining action plans. Specifically, the abstract nature of translating vague architectural uncertainties into testable hypotheses proved to be a cognitive bottleneck. The study highlighted that teams struggled with mapping the scenarios with risk and properly articulating the necessary technical interventions.

Furthermore, without structured guidance, the execution of the framework often

defaulted back to the most experienced members of the team, reinforcing the very centralization the method aims to dismantle. The feedback from practitioners was explicit regarding the solution to this problem: there is a strong need for tool support to manage hypotheses and execute action plans, which could lessen team dependency on architects. Without effective tools to aid architects—suggesting patterns and supporting decision-making—the dependency on seasoned experts remains a bottleneck, perpetuating the centralization of architectural authority.

3.1.2 Technological Intervention as a Scaffolding Mechanism

This capstone project posits that the dissonance identified in Chapter 2 — where the desire for decentralized autonomy conflicts with the reliance on centralized tacit knowledge—cannot be resolved through process changes alone. It requires a technological intervention. The development of a dedicated software tool aims to operationalize the ArchHypo framework.

By embedding the methodological rules, assessment scales, and technical patterns directly into a software interface, the tool provides the necessary structural scaffolding for less experienced practitioners. It intends to allow teams to explicitly document and manage architectural hypotheses, assess them based on impact and uncertainty, and define and track technical action plans. Consequently, the tool acts as a mechanism for knowledge distribution, enabling the decentralization of high-quality architectural decision-making.

3.2 HypoStage and the Backstage Platform Strategy

To address the identified needs, this project introduces **HypoStage**, a software solution designed to integrate architectural hypothesis management into the daily workflow of engineering teams.

3.2.1 The HypoStage Solution

HypoStage is defined as an ArchHypo plugin for Backstage. It is not merely a documentation repository but an active management tool that integrates architectural hypothesis management into the organization's IDP, enabling teams to document, track, and validate architectural decisions effectively. The tool is designed to guide users through the lifecycle of a hypothesis, from elicitation to validation or refutation, ensuring that uncertainty is treated as a first-class citizen in the software delivery process.

A key design principle of HypoStage is its capability to seamlessly integrate with Backstage's catalog and entity system. This ensures that architectural hypotheses are not abstract entities but are directly linked to the software components, APIs, and resources they affect, promoting traceability and context-aware decision-making.

3.2.2 Rationale for the Backstage Ecosystem

The decision to implement HypoStage as a plugin for **Backstage**—an open-source IDP—is strategic. To bridge the gap identified in the literature, researchers have suggested

that plugins of existing tools could introduce support for hypotheses management. By integrating with project management platforms or developer portals, a tool can facilitate seamless adoption into diverse development workflows.

Backstage was selected as the host platform because it serves as a centralized hub for software development teams, aggregating infrastructure, services, and documentation. By placing hypothesis management within the IDP, HypoStage ensures that architectural decision-making occurs in the same environment where development happens. This visibility is crucial for decentralization, as it democratizes access to architectural rationale.

3.3 Functional Requirements and Implementation

HypoStage implements specific functional requirements derived from the ArchHypo framework to support the complete lifecycle of hypothesis engineering. These functionalities cover documentation, assessment, planning, and visualization.

3.3.1 Explicit Hypothesis Management and Tracking

The core functionality of HypoStage is Hypothesis Management, which allows users to create, edit, and track architectural hypotheses with detailed metadata. The system enforces a structured format for elicitation, requiring clear statements that define the uncertainty. The implementation provides a robust form interface that captures essential data points such as the Hypothesis Statement, Source Type (e.g., Requirements, Solution), and associated Entity References from the catalog.

The process of creating a hypothesis in HypoStage follows a step-by-step workflow, as illustrated in Figures 3.1, 3.2, and 3.3. Users begin by accessing the hypothesis creation form, where they can input the hypothesis statement and select relevant metadata. The interface guides users through each step, ensuring that all necessary information is captured before the hypothesis is created.

Furthermore, the tool supports Status Tracking, allowing teams to monitor hypothesis lifecycle from creation to validation. The system supports multiple statuses for hypotheses, such as Open, In Review, Validated, Discarded, and Trigger-Fired. This explicit tracking ensures that architectural uncertainties are not forgotten but are actively managed until resolution.

Once hypotheses are created, users can view them in a comprehensive list view, as shown in Figure 3.4, which provides an overview of all hypotheses and their current status. Clicking on a specific hypothesis opens its detailed page, illustrated in Figure 3.5, where users can view complete information, edit details, and track the hypothesis lifecycle.

3.3.2 Uncertainty and Impact Assessment

A critical component of the ArchHypo framework is the quantitative and qualitative assessment of hypotheses. HypoStage implements Uncertainty Assessment functionality to evaluate hypothesis uncertainty using Likert scale ratings.

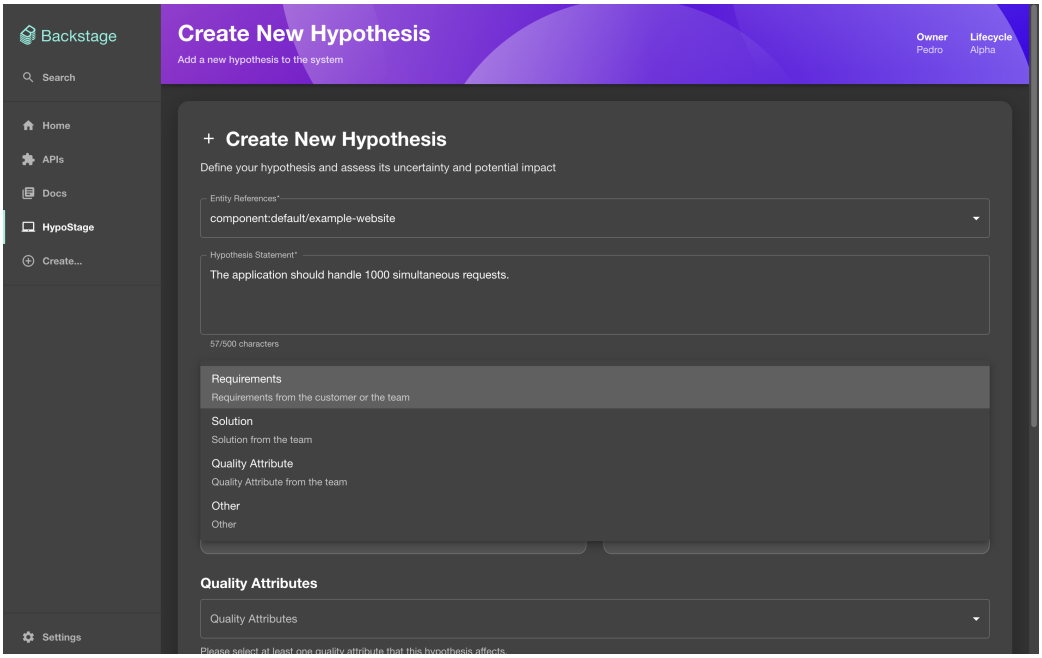

Figure 3.1: *Step 1: Accessing the hypothesis creation form in HypoStage*

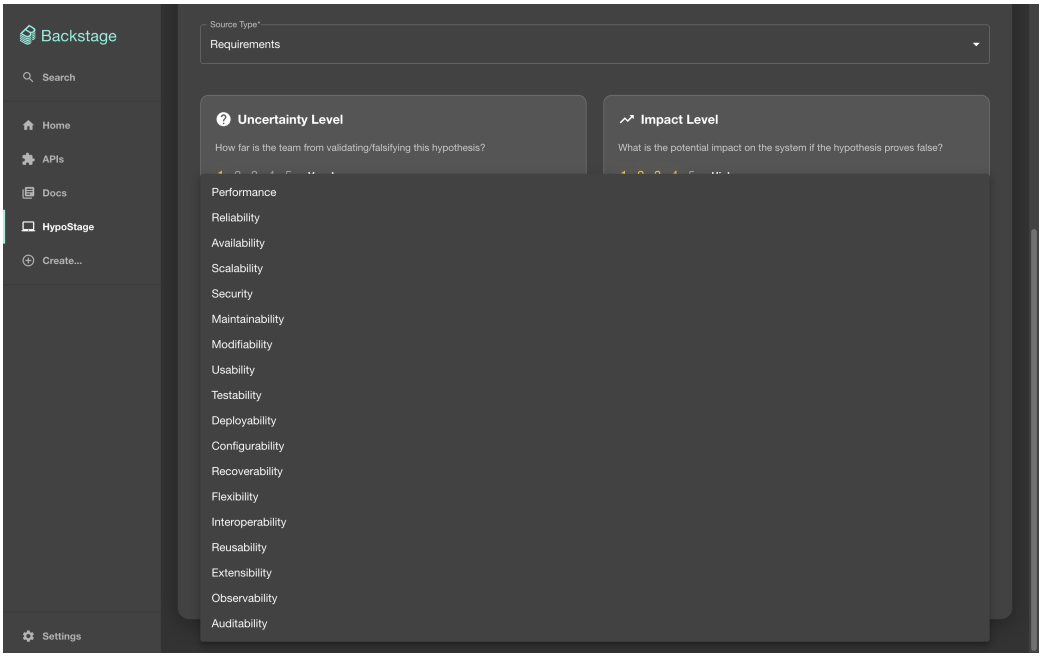

Figure 3.2: *Step 2: Filling in the hypothesis details and metadata*

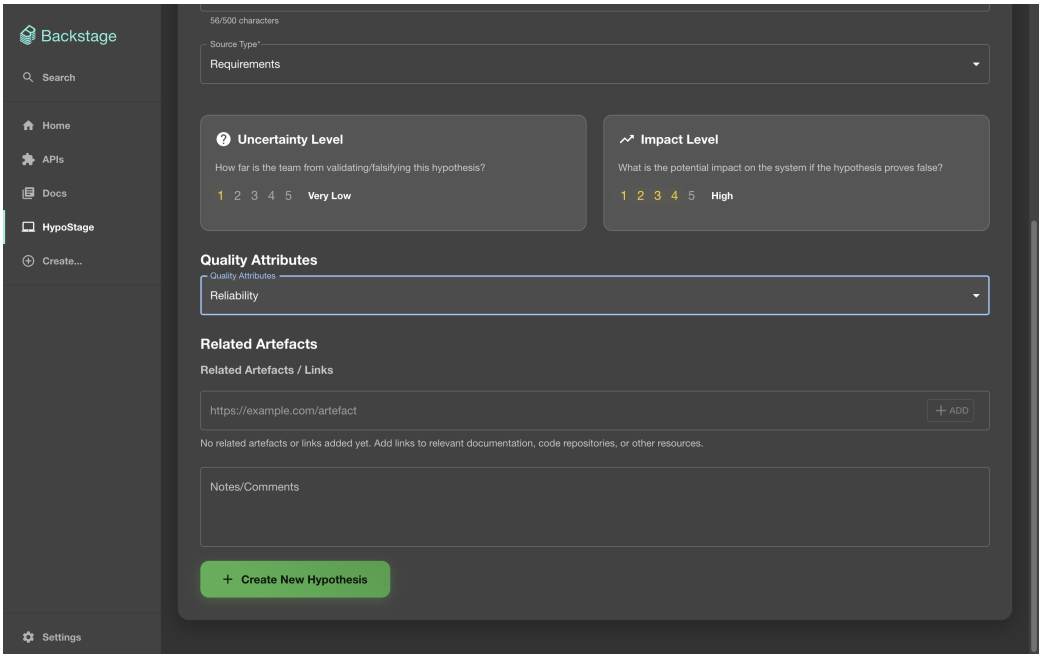

Figure 3.3: *Step 3: Completing the hypothesis creation process*

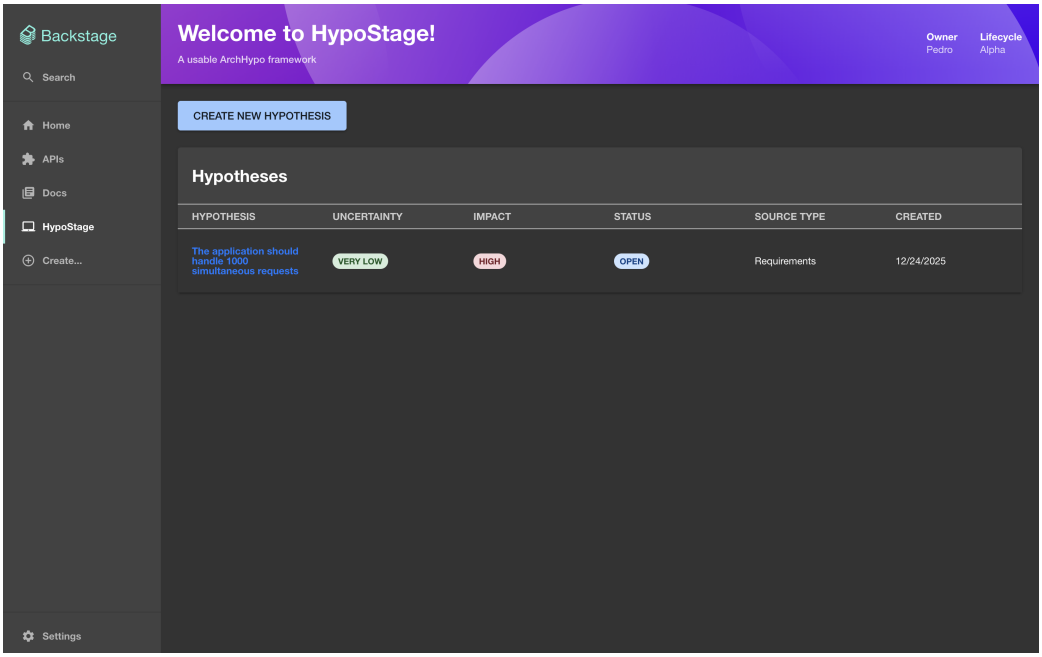

Figure 3.4: *The list view showing all created hypotheses in HypoStage*

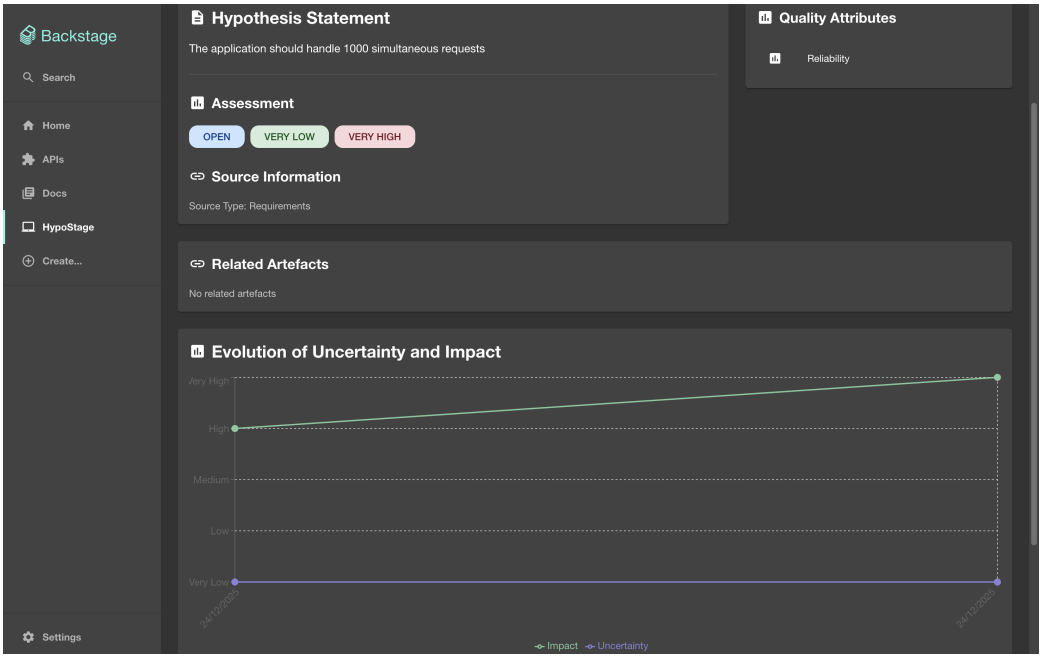

Figure 3.5: *A detailed view of a hypothesis page in HypoStage*

The interface requires users to provide an Uncertainty Rating and an Impact Rating, both utilizing a 1-5 scale. These ratings map to values ranging from Very Low (1) to Very High (5). This structured assessment forces teams to critically evaluate how far they are from validating a hypothesis and what the potential consequences of failure are, directly addressing the difficulty participants faced in mapping risks in manual implementations.

3.3.3 Technical Planning and Quality Correlation

To move from assessment to action, HypoStage provides Technical Planning capabilities. The tool allows teams to create and manage technical planning items linked to hypotheses. Users can define specific actions—such as Architectural Spike, Tracer Bullet, or Prototype — assigned to specific entities with target dates and expected outcomes. This directly supports the objective to define and track technical action plans associated with each hypothesis.

The creation of a technical plan follows a structured process, as demonstrated in Figures 3.6 and 3.7. Users can initiate the creation of a technical plan from within a hypothesis page, where they are guided through defining the action type, target entities, and expected outcomes. Once created, technical plans are displayed within the hypothesis page, as shown in Figure 3.8, providing visibility into all planned actions associated with a given hypothesis.

Additionally, the tool implements Quality Attributes Tracking, enabling users to associate hypotheses with specific quality attributes. The system supports a comprehensive list of attributes, such as Performance, Security, Scalability, and Maintainability. This feature ensures that architectural decisions are explicitly linked to the non-functional requirements they impact, fostering a quality-driven architectural culture.

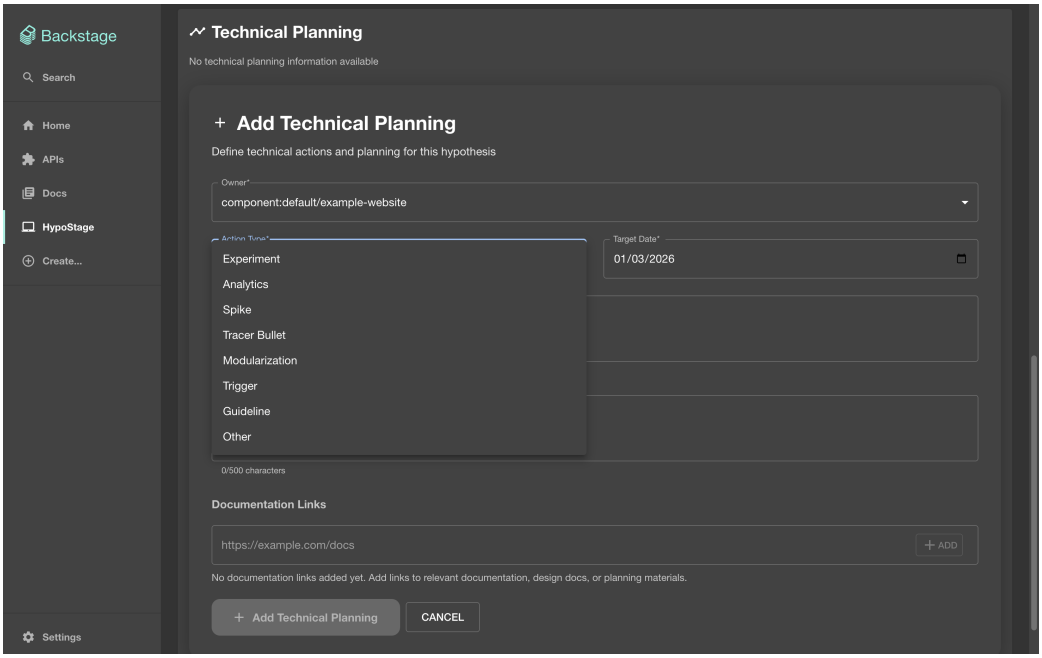

Figure 3.6: *Step 1: Initiating the creation of a technical plan in HypoStage*

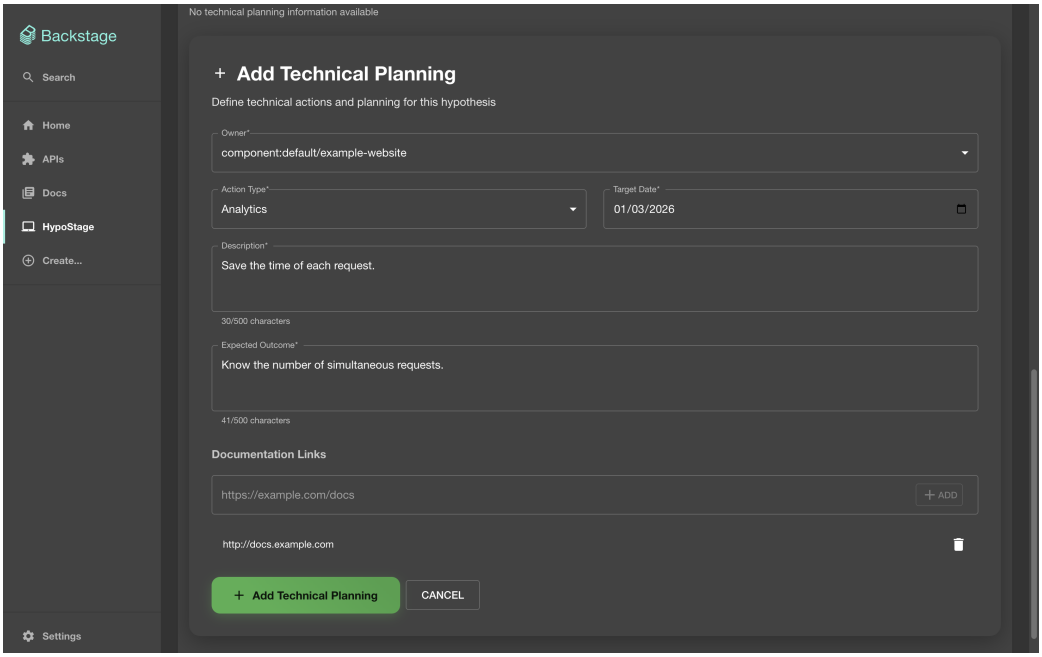

Figure 3.7: *Step 2: Defining the technical plan details and action items*

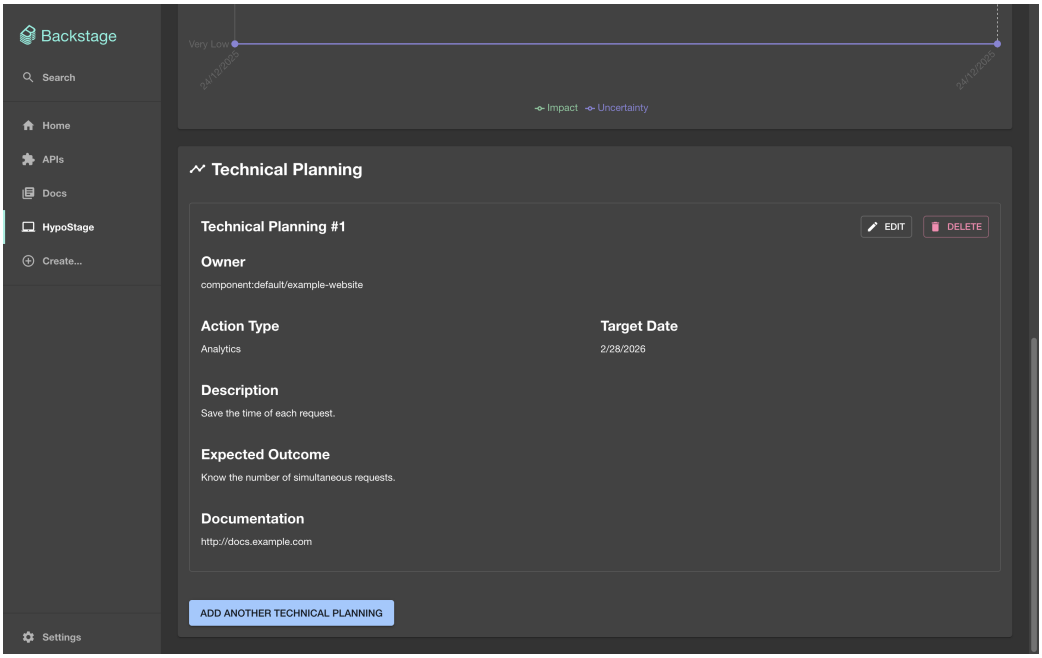

Figure 3.8: *Technical plans displayed within a hypothesis page after creation*

3.3.4 Visualization and Evidentiary Support

To support decision-making over time, HypoStage includes Visualization features to track hypothesis evolution and validation status through interactive charts. The system renders a temporal view of how uncertainty and impact ratings change as technical plans are executed, providing empirical evidence of risk reduction.

Finally, to ground decisions in facts rather than intuition, the tool supports the inclusion of Evidence URLs. The system allows teams to add supporting documentation links, such as external test results, POC repositories, or vendor documentation, thereby ensuring that the validation of a hypothesis is auditable and transparent.

3.4 Software Engineering Architecture

The engineering of HypoStage follows modern software development practices, utilizing a decoupled architecture that aligns with the Backstage plugin ecosystem. The solution is divided into two main packages: the frontend plugin and the backend plugin.

3.4.1 Component Architecture

The system architecture is composed of distinct frontend components and backend services.

Frontend Architecture

The frontend is built using React and integrates with the Backstage core API.

Backend Architecture

The backend logic is encapsulated in the backend package.

- HypothesisService: The core service for hypothesis management. As seen in this service manages business logic and data persistence. It utilizes database transactions to ensure data integrity when creating hypotheses and logging associated lifecycle events simultaneously.
- Router: The router module uses express-promise-router to define RESTful endpoints, handling request validation via Zod schemas before invoking the service layer.
- Persistence: The system uses Knex.js for database interactions, including database migrations for schema setup which define the structure for hypothesis, technicalPlanning, and hypothesisEvents tables.

3.4.2 Design for Integration and Modularity

The project adheres to the goal of ensuring modularity and integrability. The plugin is registered within the Backstage backend system using the createBackendPlugin factory, allowing it to inject dependencies such as logger, database, httpAuth, and catalogService.

This dependency injection model is crucial for the tool's operation as a socio-technical enabler. For instance, the HypothesisService interacts with the CatalogService to fetch entity references, ensuring that hypotheses are tightly coupled to the actual software components registered in the organization's ecosystem. This design allows HypoStage to be used as a standalone application (in a dev environment) or integrated with existing developer portals, fulfilling the requirement for seamless adoption.

3.5 Open Source Commitment and Documentation

Recognizing that the challenges of architectural decision-making are universal, HypoStage is positioned as a contribution to the wider software engineering community.

3.5.1 Licensing and Distribution

The project is distributed as open-source software under the LGPL-3.0 license (GNU Lesser General Public License v3.0). This licensing model was chosen to balance the freedom of use with the requirement that modifications to the plugin library itself remain open source, thereby encouraging community contribution back to the core tool while allowing integration into proprietary Backstage instances.

3.5.2 Documentation for Adoption

To mitigate the learning curve not just of the method, but of the tool itself, comprehensive documentation has been developed. The README.md serves as the primary entry point, detailing Installation, Configuration, and Usage guides. It explains how to configure the frontend routes and backend plugins, ensuring that organizations can adopt the tool

with minimal friction. By providing clear API Reference and Features descriptions, the project aims to lower the barrier to entry, facilitating the shift from centralized, tacit architectural management to a transparent, decentralized, and hypothesis-driven approach.

Chapter 4

Applying HypoStage to the Catch Solve Start-up

This chapter demonstrates the practical application of the HypoStage tool by revisiting the "Catch Solve" case study presented by Silva, Melegati, Wang, *et al.*, 2024. In the original study, the ArchHypo technique was applied manually through meetings and interviews to manage architectural uncertainty in a web testing start-up. This chapter reconstructs that experience, illustrating how HypoStage would be used to digitize, manage, and track the same architectural hypotheses, assessments, and technical plans described in the literature.

By mapping the real-world decisions faced by Catch Solve to the specific functional capabilities of HypoStage detailed in Chapter 3, we provide a concrete walkthrough of the tool's workflow from elicitation to resolution.

4.1 System Context and Entity Registration

Before managing hypotheses, the software system under analysis must be contextually defined. As described in Chapter 3, HypoStage integrates with the Backstage Catalog to link hypotheses directly to software entities.

In the context of Catch Solve, the primary system is a platform that offers testing and monitoring services for web applications. The first step in the HypoStage workflow is navigating to the Catch Solve Platform entity within the developer portal. By anchoring the hypothesis to this entity, all subsequent architectural decisions remain traceable to the specific component they affect.

4.2 Phase 1: Hypothesis Elicitation

The first phase involves capturing the architectural uncertainties identified by the development team. In the manual study, this was done via a meeting with the technical

lead. In HypoStage, this is achieved using the Hypothesis Management interface to create structured records.

4.2.1 Defining the Availability Hypothesis

One of the initial concerns raised by Catch Solve was that "The lack of redundancy can cause problems with application availability for the customers".

Using HypoStage, a user would create a new hypothesis with the following structured data:

Statement: The lack of redundancy can cause problems with application availability for the customers.

Quality Attribute: Availability.

Source Type: Solution (referring to the current architectural limitation).

Status: Open.

4.2.2 Defining the Scalability Hypothesis

A second major concern was the manual creation of tests, which limited the start-up's ability to scale. The team hypothesized that "Test templates can be used to generate tests for different applications".

In HypoStage, this entry would be recorded as:

Statement: Test templates can be used to generate tests for different applications.

Quality Attribute: Reusability.

Source Type: Requirement (focusing on the need to optimize the test creation process).

Status: Open.

4.3 Phase 2: Uncertainty and Impact Assessment

Once the hypotheses are documented, HypoStage requires a quantitative assessment to prioritize them. The tool utilizes a 5-point Likert scale for both Uncertainty and Impact.

4.3.1 Assessing Availability

In the case study, the team realized that implementing redundancy was not immediately critical because the application was not mission-critical and customers had not complained. Furthermore, the team already knew how to implement redundancy if needed (low uncertainty).

HypoStage Input:

Uncertainty Rating: 1 (Very Low) — Rationale: The solution is known.

Impact Rating: 2 (Low) — Rationale: Current customer base tolerates occasional downtime.

4.3.2 Assessing Test Templates

Conversely, the "Test Templates" hypothesis presented a significant challenge. The founder did not know how to implement parameterized tests (high uncertainty) and recognized that failure here would affect the core execution component (high impact).

HypoStage Input:

Uncertainty Rating: 5 (Very High) — Rationale: Implementation path is unknown.

Impact Rating: 5 (Very High) — Rationale: Affects core business scalability.

The Visualization feature of HypoStage would immediately highlight this contrast.

4.4 Phase 3: Technical Planning

HypoStage moves beyond simple documentation by allowing teams to attach executable Technical Plans to each hypothesis.

4.4.1 Planning for Availability (The Trigger)

Since the availability hypothesis had low impact and uncertainty, the decision was to postpone the architectural change. The plan was to monitor the system and revisit the decision only if the customer base grew.

In HypoStage, a Technical Planning Item is added to this hypothesis:

Action Type: Architectural Trigger.

Description: Monitor unavailability reports. Revisit redundancy architecture if new customer count exceeds threshold.

4.4.2 Planning for Test Templates (The Spike)

For the high-risk "Test Templates" hypothesis, the team needed to reduce uncertainty through experimentation. The study describes using a "Tracer Bullet" to create a reusable template instance and an "Architecture Spike" to investigate existing test suites.

In HypoStage, two planning items are created:

Action Type: Tracer Bullet.

Description: "Create a single reusable test template instance to valid integration with the current runner."

Action Type: Architectural Spike.

Description: "Analyze existing customer test suites to identify recurrent verification patterns."

4.5 Phase 4: Execution, Evidence, and Evolution

The power of HypoStage lies in its lifecycle tracking. As time progresses, the team returns to the tool to update the status based on the results of their technical plans.

4.5.1 Ten Months Later: Re-evaluating Test Templates

The case study reported that after 10 months, students had implemented proofs of concept (PoC), and a feature to check for broken URLs was successfully introduced.

Using HypoStage, the team updates the hypothesis to reflect this progress:

Evidence URLs: The user adds links to the student PoC repositories and the pull request for the "Broken URL" feature.

Re-Assessment:

Uncertainty: Downgraded from 5 (Very High) to 2 (Low).

Impact: Downgraded from 5 (Very High) to 2 (Low).

Status: Changed from "Open" to "Validated".

4.5.2 Handling the Availability Trigger

During the same period, the availability hypothesis remained stable. However, the team implemented some database redundancy for maintainability reasons, which had the side effect of improving availability.

In HypoStage, this evolution is recorded by adding a comment to the hypothesis history or linking a new "Maintainability" hypothesis that references the original "Availability" concern. This captures the "side effect" nature of the architectural evolution described in the study.

4.6 Conclusion

This walkthrough illustrates how HypoStage transforms the abstract ArchHypo framework into a concrete, manageable workflow. By using the Catch Solve data, we demonstrated how the tool supports the full lifecycle of architectural decision-making:

Elicitation: Converting vague concerns into structured Hypothesis Entities linked to the Backstage Catalog.

Assessment: Using the Likert Scale Interface to visually distinguish between "postponable" decisions (Availability) and "critical" investigations (Test Templates).

Planning: Assigning specific Technical Plans (Triggers vs. Spikes) to operationalize the response to uncertainty.

Tracking: Using Evidence URLs and Status Updates to provide an audit trail of how uncertainty was reduced over time.

While the original study relied on periodic meetings and manual documentation, HypoStage enables this process to occur asynchronously and continuously within the developer's native environment, ensuring that architectural knowledge remains visible, accessible, and actionable.

Chapter 5

Conclusion: Contributions and Future Directions

5.1 Summary of Contributions: Filling the Tooling Void

This capstone project addresses a fundamental operational void in modern software architecture: the critical absence of specialized tooling to manage architectural uncertainty. While the theoretical foundations for decentralized decision-making exist in frameworks like ArchHypo, their practical application has been stifled by a reliance on manual, high-friction processes that development teams struggle to sustain.

The primary contribution of this work is the delivery of the missing technological link: **HypoStage**.

HypoStage is a purpose-built software artifact designed to operationalize Hypothesis Engineering. It transforms the abstract principles of architectural uncertainty—previously managed through ad-hoc documentation or the tacit knowledge of senior architects—into a concrete, structured digital workflow. By automating the lifecycle of hypothesis elicitation, assessment, and technical planning, the tool removes the administrative burden that has historically blocked the adoption of evidence-based architectural methods.

Recognizing that modern engineering organizations are consolidating their operations into Internal Developer Portals (IDPs), HypoStage was not built in isolation but as a fully integrated plugin for **Backstage**. This strategic architectural choice ensures that hypothesis management is embedded directly into the developer's "Golden Path." By placing the tool where developers already manage their services and deployments, HypoStage bridges the gap between architectural intent and daily engineering execution, providing the necessary scaffolding for teams to practice continuous, decentralized architecture effectively.

5.2 Results: Insights Empowered by the Tool

The development of HypoStage is grounded in the proven empirical benefits of the ArchHypo technique. The tool is designed not merely to record data, but to amplify and sustain the positive outcomes observed in industrial applications of the methodology. By facilitating the management of architectural hypotheses, the tool unlocks several critical advantages for software development projects.

Support for Prioritization and Risk Management

One of the most significant impacts of applying this methodology is the enhancement of risk visibility. In complex projects, architectural risks are often opaque or implicit. The structured elicitation and assessment of hypotheses supported by the tool allow for a clearer visualization of the risks that could threaten the final delivery of the project.

By making these risks explicit and quantifiable through impact and uncertainty scores, teams can prioritize their efforts more effectively, focusing on the most critical uncertainties that could jeopardize project success.

Sustainable Task Distribution and Planning

The methodology provides a robust framework for managing the temporal dimension of architecture. Rather than treating architecture as a monolithic upfront phase, the approach supported by HypoStage enables a structured approach to dividing the architectural work through iterations. This allows for allocating and scheduling architectural tasks throughout project iterations, ensuring that architectural evolution is continuous and integrated with feature development. This distribution prevents the accumulation of unmanaged technical debt and ensures that architectural work is prioritized alongside functional requirements.

Sustainable Development and Quality

The systematic management of uncertainty contributes directly to the sustainability of the development process. Empirical evidence suggests that the adoption of this approach benefits projects by providing predictability, security, transparency, and a sustainable pace. Furthermore, the technique has been shown to contribute significantly to decision-making efficiency. By reducing the chaos associated with unforeseen architectural blockers, teams can maintain a steady velocity and deliver higher-quality software that meets both functional and non-functional requirements.

Reduction of Upfront Design

A core tenet of modern agile architecture is the avoidance of heavy upfront design, which often leads to waste and rigidity. The ArchHypo technique, operationalized by the tool, allows projects to avoid an upfront architectural design. It achieves this by providing a safety net that supports the strategic postponement of decisions while addressing their potential impact. By identifying which decisions can be safely delayed and monitoring their associated risks, teams can maintain agility and keep their options open until the "Most Responsible Moment" arises.

Informed Decision-Making

Finally, the ultimate goal of the tool is to elevate the quality of architectural decisions. By moving away from intuition-based choices, the methodology ensures that decisions are grounded in evidence and analysis. Teams that have adopted this approach recognized that the management of hypotheses compels the team to base decisions on information rather than guesses. HypoStage facilitates this by tracking the results of experiments, spikes, and analytics, ensuring that when a decision is finally made, it is backed by data.

5.3 Future Steps: Validation and Research Evolution

While the development of HypoStage represents a significant step towards resolving the challenges of decentralized architectural decision-making, it marks the beginning of a new phase of research and validation. The path forward involves rigorous empirical testing of the tool, the refinement of the underlying methodologies, and the continued expansion of the pattern language that supports ArchHypo.

Empirical Validation in Diverse Contexts

To fully understand the efficacy and generalizability of HypoStage, it is essential to conduct extensive empirical studies. Future work must evaluate the adoption of ArchHypo in companies and development teams with different backgrounds and characteristics. This includes varying team sizes, domains, and maturity levels. Furthermore, researchers must apply these patterns in other projects to study their broader impact, address challenges, and refine solutions to improve their adoption and effectiveness. These studies should aim to quantify the reduction in decision-making bottlenecks and the improvement in architectural quality.

User Acceptance Evaluation

The success of any developer tool depends on its acceptance by the practitioner community. To systematically evaluate how engineering teams perceive and interact with HypoStage, future interview and survey studies should be designed to identify the factors that influence the adoption of the tool—such as performance expectancy, effort expectancy, and social influence. Gathering this data will help refine both the user experience and the feature set.

Tool Improvement and Learning Curve Reduction

A primary motivation for HypoStage was to mitigate the steep learning curve associated with ArchHypo. Consequently, continuous investigation into tools and methodologies to improve the implementation of ArchHypo represents an important direction for research. Future iterations of the tool should focus on intelligent features to further reduce this curve. This could include AI-driven recommendation systems that suggest potential hypotheses based on project characteristics or automated generation of technical plans based on historical data.

Evolution of the Framework via New Patterns

The theoretical foundation of ArchHypo must also continue to evolve. Future research should focus on identifying and documenting patterns for various types of hypotheses. As the software landscape changes, new categories of uncertainty emerge. There is a specific need for the exploration of common uncertainties related to specific quality attributes, such as sustainability and usability. Documenting these patterns will enrich the knowledge base available to users of HypoStage, allowing them to leverage collective industry knowledge when formulating their own hypotheses.

Research on Process Guidelines

Finally, the integration of ArchHypo into the broader software development lifecycle remains a fertile area for research. The empirical study noted that the introduction of specific process guidelines was a highly effective strategy for managing uncertainty. Therefore, future studies can also investigate how these guidelines can be adopted as an approach to dealing with uncertainty. Understanding how to best weave hypothesis engineering into Agile, DevOps, and other process methodologies will be crucial for the seamless adoption of decentralized architectural decision-making.

References

[Foote and Yoder 2003] Brian Foote and Joseph Yoder. "Big ball of mud" (2003) (cit. on p. 2).

[Paris *et al.* 2024] André Paris, Eduardo Guerra, Fabio Silveira, and Kelson Silva. "Patterns for small adjustments in the development process to deal with architectural uncertainties". In: *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices*. EuroPLoP '24. New York, NY, USA: Association for Computing Machinery, 2024. ISBN: 9798400716836. DOI: 10.1145/3698322.3698335. URL: https://doi.org/10.1145/3698322.3698335.

[Perry and Wolf 1992] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the study of software architecture". *SIGSOFT Softw. Eng. Notes* 17.4 (Oct. 1992), pp. 40–52. ISSN: 0163-5948. DOI: 10.1145/141874.141884. URL: https://doi.org/10.1145/141874.141884 (cit. on p. 1).

[Petersen *et al.* 2015] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. "Guidelines for conducting systematic mapping studies in software engineering: an update". *Information and Software Technology* 64 (2015), pp. 1–18. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.03.007. URL: https://www.sciencedirect.com/science/article/pii/S0950584915000646.

[Silva, Adolfo, *et al.* 2024] Kelson Silva, Luciane Baratto Adolfo, *et al.* "Patterns for using hypothesis engineering to manage architectural uncertainties". In: *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices*. EuroPLoP '24. New York, NY, USA: Association for Computing Machinery, 2024. ISBN: 9798400716836. DOI: 10.1145/3698322.3698333. URL: https://doi.org/10.1145/3698322.3698333.

[Silva, Melegati, Silveira, *et al.* 2025] Kelson Silva, Jorge Melegati, Fabio Silveira, *et al.* "Archhypo: managing software architecture uncertainty using hypotheses engineering". *IEEE Trans. Softw. Eng.* 51.2 (Feb. 2025), pp. 430–448. ISSN: 0098-5589. DOI: 10.1109/TSE.2024.3520477. URL: https://doi.org/10.1109/TSE.2024.3520477 (cit. on pp. 1, 11, 13).

[Silva, Melegati, Wang, *et al.* 2024] Kelson Silva, Jorge Melegati, Xiaofeng Wang, Mauricio Ferreira, and Eduardo Guerra. "Using hypotheses to manage technical uncertainty and architecture evolution in a software start-up". *IEEE Software* 41.4 (2024), pp. 7–13. doi: 10.1109/MS.2024.3383628 (cit. on p. 23).

[Souza *et al.* 2019] Eric Souza, Ana Moreira, and Miguel Goulão. "Deriving architectural models from requirements specifications: a systematic mapping study". *Information and Software Technology* 109 (2019), pp. 26–39. issn: 0950-5849. doi: https://doi.org/10.1016/j.infsof.2019.01.004. url: https://www.sciencedirect.com/science/article/pii/S0950584919300035 (cit. on pp. 1, 6).

[Wirfs-Brock *et al.* 2015] Rebecca Wirfs-Brock, Joseph Yoder, and Eduardo Guerra. "Patterns to develop and evolve architecture during an agile software project". In: *Proceedings of the 22nd Conference on Pattern Languages of Programs*. PLoP '15. Pittsburgh, Pennsylvania: The Hillside Group, 2015. isbn: 9781941652039.