A Software Architecture to Support Difficulty Adaptation in Puzzle Game Level Design

Francisco Wendel Almeida Cavalcante
PPGIA & GIRA Lab, Universidade de Fortaleza
Fortaleza, Brazil
wendelcavalcante@edu.unifor.br

Maria Andréia Formico Rodrigues
PPGIA & GIRA Lab, Universidade de Fortaleza
Fortaleza, Brazil
mafr@unifor.br

Abstract

Difficulty adaptation can maintain an appropriate challenge by adjusting gameplay or content to player performance. However, adaptive puzzle generation often couples player modelling, adaptation policies, generation algorithms, difficulty assessment, and runtime integration. This paper presents a modular software architecture separating integration, orchestration, player modelling, adaptation, persistence, configurable generation pipelines, and domain-specific plugins. Pipeline stages for generation, validation, difficulty assessment, ranking, and selection are replaceable. The architecture supports stateful and stateless execution, runtime level selection, and offline candidate analysis. A Python prototype using FastAPI, SQLite, and plugins for Sudoku, Sokoban, and Fifteen Puzzle demonstrates the same workflow across distinct generation strategies, validation requirements, and difficulty metrics. The results provide initial evidence of feasibility, modularity, and extensibility; player experience and adaptation effectiveness remain future work.

Keywords

Software architecture, Difficulty adaptation, Puzzle games, Level design

1 Introduction

Maintaining an appropriate level of challenge is a central concern in game design, since challenge contributes to player engagement and the overall quality of the gameplay experience [22].

When a game is substantially easier or harder than the player's current ability, the resulting experience may become monotonous or frustrating. Dynamic Difficulty Adjustment (DDA) addresses this problem by modifying game parameters or content according to information about player behaviour and performance. Existing approaches range from predefined rules to probabilistic models, optimization methods, control theory, and machine-learning techniques [19].

Procedural Content Generation (PCG) enables game content to be created algorithmically and has been applied to levels, puzzles, rules, and other game elements [24]. In puzzle games, generation may involve constructive, generate-and-test, search-based, or reverse-generation methods, while difficulty may be estimated through structural properties, solver effort, search statistics, or player-centred measures [8, 18]. Experience-Driven PCG further connects these processes by using computational models of player experience to guide content generation toward desired experience outcomes [30].

Despite these advances, implementing adaptive puzzle generation remains a software-engineering challenge. A complete solution must collect and interpret player telemetry, maintain a player state, apply an adaptation policy, configure a suitable content-generation process, assess candidate difficulty, select an appropriate level, persist relevant data, and communicate with a game runtime. When these responsibilities are embedded in a particular game, engine, generation algorithm, or player model, the resulting implementation becomes difficult to extend, compare, and reuse across different puzzle domains. Prior work connects player modelling, adaptation, and procedural content generation [16, 27, 30]. However, existing approaches do not explicitly organize level generation, validation, difficulty assessment, candidate selection, persistence, and runtime integration as independent variation points.

This paper addresses this integration problem by proposing a modular software architecture for difficulty adaptation in puzzle game level design. Rather than prescribing a specific DDA algorithm, puzzle generator, or difficulty metric, the architecture defines explicit responsibilities and interfaces for runtime integration, workflow orchestration, player modelling, design goals, adaptation decisions, persistence, generation pipelines, and domain-specific plugins. Generation is organized as a configurable pipeline whose stages may generate, validate, assess, rank, and select candidate levels. This organization allows different methods to be combined without redesigning the complete adaptive workflow. The architecture supports both runtime and experimental use. During gameplay, it can process telemetry from a previous attempt, update the player model, determine a target difficulty, and return a selected level. For offline experimentation, it can preserve and return all generated candidates together with their validation reports, difficulty assessments, selection scores, and generation metadata. It also supports both stateless execution, in which the runtime supplies the previous player state, and stateful execution backed by a replaceable persistence mechanism.

The main contributions of this work are as follows: **(1)** A layered architecture separating integration, orchestration, adaptation, persistence, generation, and domain-specific functionality; **(2)** Configurable pipelines with replaceable generation, validation, difficulty assessment, ranking, and selection stages; **(3)** Extensibility across puzzle domains, integration mechanisms, player models, adaptation policies, and execution modes; **(4)** A Python prototype using FastAPI and SQLite, supporting stateful and stateless execution and plugins for Sudoku, Sokoban and Fifteen Puzzle; and **(5)** Case studies demonstrating reuse of the same adaptive workflow across Sudoku, Sokoban, and Fifteen Puzzle.

The remainder of this paper is organized as follows. Section 2 presents the proposed architecture and its adaptive generation flow. Section 3 describes the prototype implementation. Section 4 presents the Sudoku, Sokoban, and Fifteen Puzzle case studies. Section 5 discusses the current limitations. Section 6 positions the

proposal in relation to DDA, procedural puzzle generation, difficulty estimation, and adaptive-game frameworks. Finally, Section 7 presents the conclusions and future work.

2 Proposed Architecture

The proposed architecture supports adaptive level generation across heterogeneous runtimes, generation algorithms, and adaptive dimensions. Its functional and non-functional requirements, summarized in Table 1, guide the separation of integration, orchestration, adaptation, persistence, generation, and domain-specific concerns. The architecture was developed iteratively rather than through a formal Attribute-Driven Design process, with these functional and non-functional requirements guiding the main architectural decompositions and design decisions.

Figure 1 presents the six architectural layers and their interactions. The architecture applies Adapter, Facade, Builder, and Strategy [11], together with Pipes and Filters [3], to decouple game engines, separate responsibilities, and support extensibility. In this work, modularity and extensibility are addressed primarily as architectural design properties: they are supported by explicit interfaces, replaceable components, and layer boundaries, rather than by empirical measurements of change effort or coupling.

1. Integration Layer. This layer decouples the architecture from game engines and communication mechanisms. Following the Adapter pattern, it maps runtime interactions to request and response Data Transfer Objects (DTOs), supporting HTTP, JSON files, simulations, WebSockets, and engine-specific connectors.

2. Orchestration Layer. The Adaptive Generation Orchestrator acts as an application-level Facade, coordinating telemetry processing, player modelling, adaptation, pipeline construction, execution, and response composition.

3. Adaptation Core. This core transforms player-related information and design goals into an adaptation decision. The Telemetry Collector summarizes runtime or simulated telemetry, the Player Model updates the estimated player state, the Design Goals Manager provides designer-defined objectives and constraints, and the Adaptation Engine produces an adaptation request for the generation process. These components are designed as interchangeable strategies, allowing different player modeling techniques, adaptation policies, and goal-management mechanisms to be replaced without changing the orchestration layer.

4. Persistence Layer. Following Ports and Adapters, this layer defines storage abstractions for player states, telemetry, performance observations, adaptation decisions, and generation metadata. Replaceable adapters support stateful execution without coupling the core to a specific database, while stateless execution remains available. As a result, different storage backends, such as local files, relational databases, external services, or simulation-specific repositories, can be incorporated without changing the core adaptive generation workflow.

5. Generation Core. This core builds and executes generation pipelines according to the adaptation request. The Pipeline Builder applies the Builder pattern to assemble a pipeline from the requested domain, generation strategy, and adaptation parameters. The resulting pipeline follows a Pipes and Filters style, in which each stage receives a generation context, enriches or filters candidate records, and forwards the updated context to the next stage. Since stages are composable, the same core can support different strategies, including generate-and-test, reverse generation, constructive generation, or future machine-learning-based approaches.

6. Plugins. Plugins encapsulate domain-specific functionality, such as generators, validators, difficulty assessors, and candidate selectors for a particular game domain or content type. These elements follow the Strategy pattern, since each plugin provides interchangeable implementations of the interfaces used by the generation pipeline. In the current prototype, plugins were implemented for Sudoku, Sokoban, and Fifteen Puzzle. This design allows new

Table 1: Functional and non-functional requirements guiding the proposed architecture.

ID	Requirement	Description
FR1	Telemetry ingestion	The architecture must receive runtime or simulated telemetry describing player interaction with a level.
FR2	Player modeling	The architecture must update or receive a player state representing estimated skill, engagement, frustration, and confidence.
FR3	Adaptive decision making	The architecture must transform player state and design goals into an adaptation request for level generation.
FR4	Configurable generation pipelines	The architecture must support configurable pipelines composed of generation, validation, assessment, and selection stages.
FR5	Candidate-level analysis	The architecture must allow returning all generated candidates and their metadata, supporting experimental analysis beyond runtime selection.
FR6	Adaptive data persistence	The architecture must support the persistence of player states, telemetry, observations, adaptation decisions, and generation metadata across requests when stateful execution is enabled.
FR7	Execution modes	The architecture must support both stateless execution, in which the previous player state is received and the updated state is returned, and stateful execution, in which player states are retrieved and persisted across requests.
NFR1	Engine independence	The architecture must not depend on a specific game engine, runtime, or scene representation.
NFR2	Integration independence	The architecture must support different integration mechanisms, such as HTTP APIs, file-based JSON exchange, simulations, or future adapters.
NFR3	Extensibility	The architecture must allow new domains, generation strategies, validators, difficulty assessors, selectors, and adaptive dimensions to be plugged in.
NFR4	Observability & testability	The architecture must expose adaptation and generation metadata to support debugging, reproducibility, experimentation, and isolated component testing.

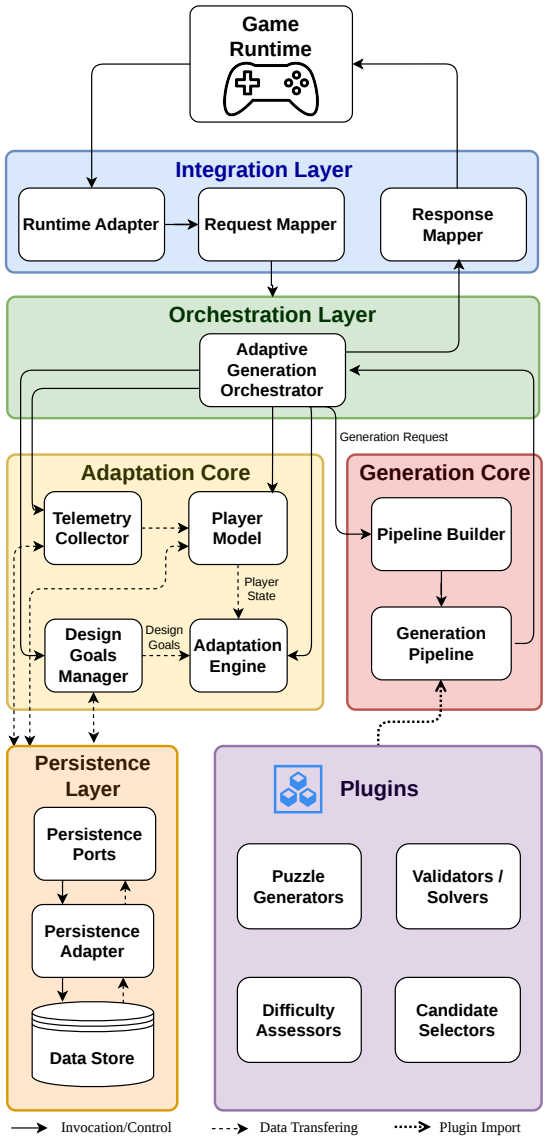

Figure 1: Overview of the proposed architecture separating runtime integration, orchestration, adaptive decision-making, persistence, generation pipelines, and domain-specific plugins. Solid lines indicate invocation/control flow, dashed lines indicate data transfer, and dotted arrows indicate plugin imports.

games, content domains, validation methods, or difficulty metrics to be added without changing the orchestration or adaptation layers.

2.1 Adaptive Generation Flow

The adaptive generation flow starts when a game runtime or simulation requests the next level. The request includes runtime information, the telemetry collected from the previous level attempt, design goals, and generation parameters. Depending on the execution mode, the request may also include the last known player state. This request is received by the Integration Layer and converted into a request DTO, which is then forwarded to the Orchestration Layer. The Adaptive Generation Orchestrator coordinates the remaining workflow. First, it converts the request DTO into internal domain objects, including a telemetry summary, design goals, and generation configuration. The player state is then resolved according to the execution mode. In stateless execution, the last player state is provided as part of the request. In stateful execution, the orchestrator retrieves the latest player state through the Persistence Layer using the player identifier and the requested domain. If no persisted state exists, an initial player state is created.

After the previous player state is resolved, the telemetry summary is processed by the player modelling component, producing an updated player state and a performance observation. The Persistence Layer may then store the received telemetry, the updated player state, and the performance observation. This allows later adaptive generation requests to reuse previous interaction data without requiring the game runtime to resend the full player state.

The updated player state and the design goals are then processed by the Adaptation Engine, which produces an adaptation decision. In the current prototype, this decision is represented mainly as a target difficulty value and associated constraints. The adaptation decision may also be persisted for later inspection, debugging, or experimental analysis. The decision is converted into an adaptation request, which is used by the Pipeline Builder to assemble a suitable generation pipeline for the requested domain and strategy.

After the pipeline is executed, the Generation Core returns a generation result containing the generated candidates, their validation and difficulty reports, and, when selection is enabled, the selected level candidate. The Orchestration Layer then composes a response DTO containing the selected level, the updated player state, the adaptation decision, and generation metadata. Finally, the Integration Layer maps this response into a format consumable by the game runtime or simulation. Therefore, the same flow supports both stateless request-response execution and stateful execution backed by persistent player and telemetry data.

2.2 Configurable Generation Pipelines

Generation pipelines consist of optional, independent stages operating on a shared context. Different stage compositions represent distinct generation strategies without requiring monolithic generators. Runtime pipelines may select the candidate closest to the adaptation target, whereas experimental pipelines may return all candidates for analyses of solvability, difficulty, cost, and diversity.

The prototype instantiates this model through multi-stage Sudoku generation, reverse Sokoban generation, and generate-and-test Fifteen Puzzle generation, showing that generation, validation, and difficulty assessment remain domain-specific and replaceable.

2.3 Data Contracts

To reduce coupling between layers, the architecture defines explicit data contracts. Runtime-specific payloads are first mapped into request DTOs by the Integration Layer. The Orchestration Layer

then instantiates internal objects such as *TelemetrySummary*, *PlayerModelState*, *DesignGoals*, and *AdaptationRequest*. The Generation Core returns a *GenerationResult*, which may contain a selected *LevelArtifact* as well as the complete set of generated *CandidateRecord* instances. This separation allows the same core abstractions to be reused across different runtimes, integration mechanisms, and experimental settings.

2.4 Runtime and Experimental Usage

Pipelines may either select a level at runtime or return all candidate records for offline analysis. Thus, the same architecture supports adaptive gameplay and empirical evaluation of generation methods.

Runtime use does not require frame-level or hard real-time execution, since generation occurs at puzzle/level boundaries. Its response time therefore depends mainly on the generation strategy and domain complexity.

3 Prototype Implementation

The architecture was implemented as a Python prototype organized into integration, orchestration, adaptation, persistence, generation, and plugin packages. Lightweight implementations exercise the complete workflow, while abstract interfaces allow generators, validators, difficulty assessors, player models, adaptation policies, persistence adapters, and integration mechanisms to be replaced. The prototype provides implementation evidence of extensibility by instantiating the same orchestration and generation infrastructure with three different puzzle domains, different generation strategies, different validation requirements, and different difficulty assessment methods.

The implementation uses explicit data structures for the main artifacts exchanged between layers. Runtime or simulation requests use DTOs, while internal components operate on domain objects such as telemetry summaries, player states, design goals, adaptation requests, level artifacts, and generation results. This keeps integration mechanisms thin and core components independent of external protocols or engine-specific formats.

The prototype supports stateless and stateful execution. In stateless mode, the runtime sends the latest player state with each request. In stateful mode, the orchestration layer retrieves and stores player states through a persistence port implemented with SQLite. States are stored per player-domain pair to prevent skill estimates from being shared across unrelated puzzle domains. In addition, the prototype also implements a lightweight probabilistic player-model updater as one DDA strategy. Given the current skill estimate and previous level difficulty, it estimates the expected success probability. Telemetry then updates the skill estimate from the difference between expected and observed success, while uncertainty decreases as observations accumulate. This simple implementation remains replaceable within the Adaptation Core; dynamic scripting, reinforcement learning, bandit methods, or neural models could use the same interface.

A central part of the system is the generation pipeline infrastructure. A pipeline is composed of stages that operate over a shared generation context. Candidate levels are stored as candidate records, which may contain the generated level artifact, validation report, difficulty report, selection score, activation status, and metadata. This representation allows the pipeline to support both runtime and experimental usage. In runtime-oriented executions, a selection stage chooses a single candidate to be returned to the game. In experimental executions, the pipeline returns all generated candidates, including invalid or inactive ones, enabling analyses such as solvability rates, difficulty distributions, and generation cost.

The adaptive generation orchestrator maps request DTOs to internal objects, updates the player model, applies the adaptation policy, executes the generation pipeline, and composes the response. A thin FastAPI adapter exposes this workflow through HTTP without embedding adaptation or generation logic.

The prototype also includes a plugin structure for domain-specific implementations. Each plugin may provide generators, validators, difficulty assessors, and candidate selectors for a particular game domain. In the current implementation, plugins were developed for Sudoku, Sokoban, and Fifteen Puzzle. These plugins are not intended to define definitive pipelines for each puzzle, but to demonstrate how different generation strategies and evaluation components can be plugged into the same architectural workflow.

4 Architecture Case Studies

This section presents three end-to-end case studies that instantiate the proposed architecture in different puzzle domains. The goal is not to evaluate adaptive gameplay with human participants at this stage, but to demonstrate how runtime inputs are processed through the integration, orchestration, adaptation, and generation layers to produce an adaptive level response. Each case study follows the complete workflow from telemetry and player-state input to pipeline execution and level output.

We focus on Sudoku, Sokoban, and Fifteen Puzzle because they exercise distinct architectural variation points. Sudoku uses a multi-stage generate-and-transform pipeline with explicit uniqueness validation, Sokoban uses reverse generation with construction-based solvability, and Fifteen Puzzle uses a simpler generate-and-test pipeline. All three cases follow the same adaptive workflow while differing in their generation, validation, and difficulty-assessment strategies.

4.1 Sudoku Case Study

Sudoku is a logic-based number-placement puzzle whose generalized form is NP-complete [31] and commonly used to evaluate constraint-solving methods [26]. The Sudoku case study uses a multi-stage generation workflow. The request specifies the domain, generation strategy, candidate count, generation parameters, player state, and telemetry. The adaptation core updates the player model and computes a target difficulty, after which the pipeline generates solved grids, removes clues, validates uniqueness, assesses difficulty, and selects the candidate closest to the target. This case study demonstrates how the architecture supports pipelines in which generation and puzzle construction are separated. The generator produces complete solutions, while a subsequent stage derives playable puzzles by removing clues. The uniqueness validator and difficulty assessor remain independent stages, which means that alternative Sudoku generators, solvers, clue-removal policies, or difficulty metrics can be incorporated by replacing the corresponding components. For example, Sudoku generation could be implemented using

constraint-based methods or Wave Function Collapse [17], while difficulty could be estimated using search effort, human-inspired solving techniques, cross-entropy-based metrics, or learned predictors. Figure 2 illustrates the Sudoku pipeline, including solved-grid generation, clue removal, uniqueness validation, difficulty assessment, and candidate selection.

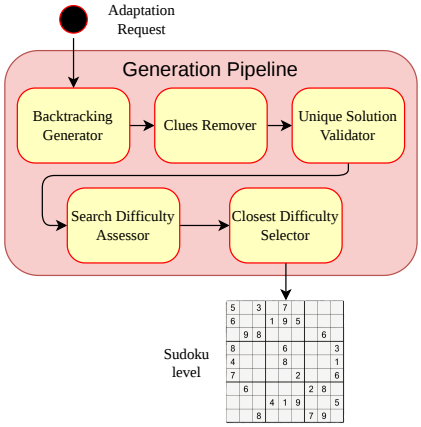

Figure 2: Sudoku generation pipeline.

4.2 Sokoban Case Study

Sokoban is a box-pushing puzzle with irreversible moves and a PSPACE-complete generalized form [7]. The case study uses reverse generation: telemetry and player state determine a target difficulty, and the pipeline constructs puzzles from solved states through reverse moves, ensuring solvability by construction. Difficulty is estimated from generation metadata, and the candidate closest to the target is selected. The case demonstrates separation between solved-grid generation, clue removal, uniqueness validation, difficulty assessment, and selection. Each stage can be replaced independently by alternative generators, solvers, removal policies, or difficulty metrics. Figure 3 presents the Sokoban pipeline, which relies on reverse generation followed by metadata-based difficulty assessment and candidate selection.

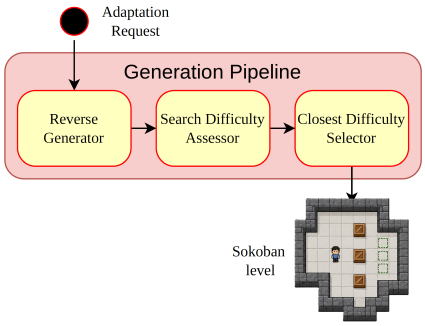

Figure 3: Sokoban generation pipeline.

4.3 Fifteen Puzzle Case Study

The Fifteen Puzzle is a classical sliding-tile benchmark in heuristic search [14]. The pipeline generates randomized boards, filters unsolvable candidates, estimates difficulty using Manhattan distance [21], and selects the candidate closest to the target difficulty. This case study represents the most direct instantiation of the architecture. Unlike Sudoku, it does not require a transformation stage such as clue removal, and unlike Sokoban, it does not rely on reverse generation to guarantee solvability by construction. Instead, it demonstrates how the architecture supports a conventional generate-and-test pipeline in which generation, validation, assessment, and selection are implemented as independent and replaceable stages. Figure 4 summarizes the pipeline used in the Fifteen Puzzle case study.

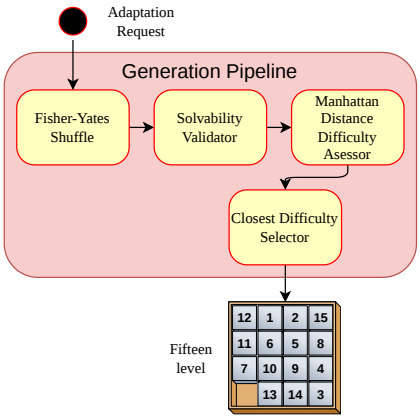

Figure 4: Fifteen Puzzle generation pipeline.

4.4 Cross-Case Summary

Table 2 summarizes the implemented case studies. The examples show that the same orchestration workflow can support heterogeneous generation strategies, validation requirements, and difficulty assessment methods. These cases demonstrate that the architecture organizes domain-specific behavior through plugins while preserving a common adaptive generation workflow.

Table 2: Puzzle case studies and instantiations.

Case	Main strategy	Architectural aspects exercised
Sudoku	Multi-stage generation	Candidate transformation, uniqueness validation, search-based difficulty assessment, closest-difficulty selection.
Sokoban	Reverse generation	Construction-based solvability, internal wall configuration, metadata-based difficulty assessment, closest-difficulty selection.
Fifteen Puzzle	Generate-and-test	Random permutation generation, solvability validation, Manhattan-distance difficulty assessment, closest-difficulty selection.

5 Limitations

This work evaluates the architecture at the implementation and use-case level, but not yet through complete adaptive puzzle games with end users. Thus, the evidence supports feasibility and modularity, but not player experience or adaptation effectiveness. Future work will address this limitation through implemented game instances, user studies, and comparisons with existing adaptation approaches.

The architectural choices may also introduce trade-offs. Stateful execution introduces persistence and consistency overhead, whereas stateless execution shifts state-management responsibility to the runtime. For simpler games with fixed adaptation and generation logic, this level of modularity may be unnecessary, and a more tightly coupled design may be preferable.

Although the prototype demonstrates extensibility across three puzzle domains, we did not quantitatively measure modularity or extensibility through metrics such as coupling, cohesion, or change effort.

6 Related Work and Positioning

Related work was organized into three areas: dynamic difficulty adjustment, puzzle level generation and difficulty estimation, and software frameworks for adaptive game design. These areas correspond to the main concerns addressed by the proposed architecture: when to adapt, how to generate or select puzzle content, and how to organize these components into a reusable pipeline.

6.1 Dynamic Difficulty Adjustment

DDA relies on information about the player to adapt game content or gameplay parameters during play. Lopes et al. [15] categorize player data sources into game telemetry, physiological data, questionnaires, bodily expressions, peripheral input, and third-person annotations. In this work, we focus on game telemetry, since the proposed architecture uses interaction measurements, such as success, solving time, mistakes, restarts, and hints, to update the player model and support adaptive generation decisions.

After defining telemetry as the primary player-data source, we considered how different DDA approaches could be incorporated into the architecture. Mortazavi et al. [19] distinguish between rule-based methods, which adjust difficulty through predefined rules, and data-driven methods, which use statistical or machine-learning techniques to model player experience and adapt game difficulty. They also discuss several families of DDA techniques, including search-based methods, control theory, probabilistic models, reinforcement learning, regression models, and neural networks. This diversity suggests that DDA should not be hard-coded as a single adaptation algorithm.

Accordingly, our architecture treats player modeling and adaptive decision making as replaceable components within the Adaptation Core. The current prototype implements a lightweight probabilistic player model updater, in which the player's skill estimate is adjusted according to the prediction error between expected and observed success. However, the same architectural interfaces could support alternative DDA strategies. For example, a developer could implement DDA as an optimization problem following Xue et al. [29], or adopt imitation and reinforcement learning approaches such as those discussed by Fuchs et al. [10]. Therefore, the proposed architecture is designed to remain agnostic with respect to the specific DDA technique used, while still providing a concrete implementation for experimentation.

In contrast to approaches centred on a particular DDA algorithm, the proposed architecture treats player modelling and adaptation policies as interchangeable strategies behind stable interfaces.

6.2 Puzzle Level Generation and Difficulty Estimation

Kegel and Haahr [8] classify puzzle generation as constructive, generate-and-test, reverse, and search-based. Our architecture supports these as interchangeable pipelines with replaceable validation, difficulty assessment, and selection stages. Babin and Katchabaw [2] further show how PCG can support adaptive game balance.

Recent works on Sudoku generation further illustrate that puzzle generation techniques continue to evolve and that a single fixed pipeline would be too restrictive. Ates and Cavdur [1] propose Sudoku puzzle generation through mathematical programming and heuristics, allowing puzzle construction to be controlled by model parameters and later combined with heuristics to address uniqueness. Saravana et al. [23] present a cognitively grounded adaptive Sudoku framework that uses human-inspired solving heuristics for difficulty estimation and real-time skill assessment. These works are not implemented in the current prototype, but they motivate one of the main architectural decisions: the Sudoku pipeline implemented in this work should be interpreted as one possible instantiation, while generation algorithms, uniqueness validators, solvers, difficulty assessors, and adaptation models remain replaceable components. A related concern is how puzzle difficulty is estimated. Valenzuela et al. [28] use search algorithm statistics, such as BFS and A* metrics [13], as proxies for maze and Sokoban difficulty, while Chen [6] relies on Shannon Information Theory [25] to introduce entropy as a Puzzle measure of difficulty. These studies motivate us to treat the difficulty assessment as a pluggable pipeline stage, enabling solver-based, search-based, data-driven, or human-centered metrics to be incorporated without changing the generation workflow.

However, these studies generally bind generation and difficulty assessment to a particular puzzle or method, whereas our architecture exposes both as replaceable pipeline stages.

6.3 Frameworks for Adaptive Game Design

Several works have approached adaptation in games from a broader framework perspective, rather than focusing only on a specific DDA algorithm, generation technique, or difficulty metric. Charles et al. [5] argue that player modelling and adaptive technologies can support player-centred game design by enabling game systems to provide experiences that better match individual players. Similarly, Zhu and Ontañón [32] discuss player-centred Artificial Intelligence for automatic game personalization, emphasizing the need to connect player models, adaptation mechanisms, and experience design goals.

Experience-Driven Procedural Content Generation (EDPCG) is another important foundation for adaptive game design. Yannakakis

and Togelius [30] introduce a framework in which procedural content generation is guided by computational models of player experience. This perspective is closely related to adaptive generation because it frames content generation as a process informed by player data and desired experience outcomes. Lopes and Bidarra [16] also propose a semantic generation framework for adaptive game worlds, emphasizing the generation of personalized content for complex and immersive environments.

More recent systems further demonstrate the value of integrating player modelling and content adaptation into reusable workflows. González-Duque et al. [12] present Fast Bayesian Content Adaptation, a domain-agnostic system that combines Bayesian player modelling with content adaptation and targets specific difficulty levels across different games. Fernandes et al. [9] propose an architecture that uses persona agents and experience metrics to evolve procedurally generated levels tailored to different player personas.

These works suggest that adaptive game design benefits from architectures that explicitly connect player data, player modelling, adaptation decisions, and content generation. The proposed architecture follows this direction, but focuses on a modular software architecture in which integration, persistence, adaptation, generation pipelines, and domain-specific plugins are separated into replaceable components. This separation allows different player modelling approaches, adaptation policies, generation strategies, difficulty assessors, and integration mechanisms to be combined without requiring the entire system to be redesigned.

Unlike prior studies that primarily propose specific adaptation algorithms, player models, or content-generation methods, our work focuses on their software-level integration across heterogeneous puzzle domains, runtimes, and experimental settings. The proposed architecture separates runtime integration, orchestration, player modelling, adaptation decisions, configurable generation pipelines, persistence, and domain-specific plugins. From a software-engineering perspective, it also specializes principles from self-adaptive systems by modelling generation, validation, difficulty assessment, selection, persistence, and game-runtime integration as independent variation points. It supports both runtime level selection and experimental analysis of allgenerated candidates. Table 3 compares the proposed architecture with representative approaches discussed in this section. The comparison focuses on the architectural concerns most closely related to this work.

Overall, the comparison highlights that the proposed architecture combines player modelling, adaptive generation, configurable pipelines, multiple puzzle domains, runtime use, and experimental analysis within a single modular software structure.

Table 3: Comparison with representative adaptive content-generation approaches.

Criterion	[30]	[16]	[12]	[9]	[5]	[32]	Ours
Player/experience model	Yes	Yes	Yes	Personas	Yes	Conceptual	**Yes**
Adaptive content generation	Yes	Yes	Yes	Yes	Yes	Conceptual	**Yes**
Multiple game domains	Conceptual	No	Yes	No	Yes	Conceptual	**Yes**
Configurable pipeline	Conceptual	Limited	No	Limited	No	No	**Yes**
Runtime adaptation	Yes	Yes	Yes	No	Yes	Conceptual	**Yes**
Experimental analysis	Yes	Limited	Yes	Yes	No	No	**Yes**

Conceptual indicates conceptual support without an explicit modular software component. *Limited* indicates approach-specific partial support.

7 Conclusion and Future Work

This paper presented a modular software architecture for difficulty adaptation in puzzle game level design. It separates runtime integration, orchestration, player modelling, adaptation, persistence, generation pipelines, and domain-specific plugins. A Python prototype with FastAPI, SQLite, and plugins for Sudoku, Sokoban, and Fifteen Puzzle demonstrated that the same workflow can support distinct generation strategies, validation mechanisms, and difficulty metrics. The results provide initial evidence of feasibility, modularity, and extensibility. Future work will integrate the architecture into playable games, evaluate it with users, compare it experimentally with alternative architectures and frameworks, compare alternative player models, adaptation policies, generation methods, and difficulty assessors, assess the effectiveness of dynamic difficulty adaptation in actual gameplay, and investigate broader AI-driven personalization of puzzle content and narratives based on player performance, preferences, and affective state [20].

Artifact Availability

The prototype and replication package are publicly archived on Zenodo, with source code maintained on GitHub [4].

Acknowledgements

The authors thank the Universidade de Fortaleza and GIRA Lab for their support. Francisco Wendel A. Cavalcante acknowledges the financial support of the Tribunal Regional Eleitoral do Ceará (TRE-CE), process no. 2024.0.000012563-0, through a doctoral scholarship. Maria Andréia F. Rodrigues acknowledges support from CNPq Grant 314776/2023-0.

References

[1] Tugce Ates and Fatih Cavdur. 2025. Sudoku puzzle generation using mathematical programming and heuristics: Puzzle construction and game development. *Expert Systems with Applications* 282 (2025), 127710. doi:10.1016/j.eswa.2025.127710

[2] Mathias Babin and Michael Katchabaw. 2025. Game Balance Through Procedural Content Generation. In *Proceedings of the 20th International Conference on the Foundations of Digital Games*. Association for Computing Machinery, Article 49, 4 pages. doi:10.1145/3723498.3723748

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Vol. 1. Wiley.

[4] Francisco Wendel Almeida Cavalcante and Maria Andréia Formico Rodrigues. 2026. Replication Package for A Software Architecture to Support Difficulty Adaptation in Puzzle Game Level Design. doi:10.5281/zenodo.22036960

[5] Darryl Charles, Michael McNeill, Moira McAlister, Michaela Black, Adrian Moore, Karl Stringer, Julian Kücklich, and Aphra Kerr. 2005. Player-centred game design: Player modelling and adaptive digital games. In *Proceedings of DiGRA 2005*

Conference: Changing Views–Worlds in Play. Digital Games Research Association: DiGRA, 285–298.

[6] Eugene You Chen Chen, Adam White, and Nathan R. Sturtevant. 2023. Entropy as a Measure of Puzzle Difficulty. In *Proceedings of the 2023 AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 19. 34–42. doi:10.1609/aiide.v19i1.27499

[7] Joseph C. Culberson. 1998. Sokoban is PSPACE-complete. In *Proceedings of the 1998 International Conference on Fun with Algorithms (FUN)*. Carleton Scientific, 65–76.

[8] Barbara De Kegel and Mads Haahr. 2020. Procedural Puzzle Generation: A Survey. *IEEE Transactions on Games* 12, 1 (2020), 21–40. doi:10.1109/TG.2019.2917792

[9] Pedro M. Fernandes, Jonathan Jørgensen, and Niels N. T. G. Poldervaart. 2021. Adapting Procedural Content Generation to Player Personas Through Evolution. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*. 01–09. doi:10.1109/SSCI50451.2021.9659880

[10] Ronja Fuchs, Robin Gieseke, and Alexander Dockhorn. 2024. Personalized dynamic difficulty adjustment imitation learning meets reinforcement learning. In *2024 IEEE Conference on Games (CoG)*. IEEE, 1–2. doi:10.1109/CoG60054.2024.10645659

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[12] Miguel González-Duque, Rasmus Berg Palm, and Sebastian Risi. 2021. Fast Game Content Adaptation Through Bayesian-Based Player Modelling. In *Proceedings of the 2021 IEEE Conference on Games (CoG)*. IEEE, 1–8. doi:10.1109/CoG52621.2021.9619018

[13] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. doi:10.1109/TSSC.1968.300136

[14] R E Korf. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27, 1 (1985), 97–109. doi:10.1016/0004-3702(85)90084-0

[15] Phil Lopes, Nuno Fachada, and Maria Fonseca. 2025. *Closing the loop: A systematic review of experience-driven game adaptation*. arXiv:2505.01351 doi:10.48550/arXiv.2505.01351 Preprint.

[16] Ricardo Lopes and Rafael Bidarra. 2011. A semantic generation framework for enabling adaptive game worlds. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/2071423.2071431

[17] Carlos Marín-Lora and Miguel Chover. 2023. A Multi-agent Sudoku Through the Wave Function Collapse. In *Multi-Agent Systems*, Vadim Malvone and Aniello Murano (Eds.). Springer Nature Switzerland, Cham, 381–395. doi:10.1007/978-3-031-43264-4_24

[18] Leonardo Gonçalves Marques, Juliana Amaral de Figueiredo, João Vitor Vieira Lira, and Maria Andréia Formico Rodrigues. 2025. A Serious Game for Bone Cancer Awareness and Education: Integrating Simulated User Feedback for Preliminary Validation. In *2025 IEEE Conference on Serious Games and Applications for Health (SeGAH)*. 1–8. doi:10.1109/SeGAH65397.2025.11168446

[19] Fatemeh Mortazavi, Hadi Moradi, and Abdol-Hossein Vahabie. 2024. Dynamic difficulty adjustment approaches in video games: a systematic literature review. *Multimedia Tools and Applications* 83, 35 (2024), 83227–83274. doi:10.1007/s11042-024-18768-x

[20] Maria Andréia Formico Rodrigues, R G Barbosa, and Jonas de A. Luz Junior. 2026. Games and Their Multiple Possible Uses in Health Care. In *Digital Healthcare: A Multidisciplinary and Comprehensive Vision*. Springer, 269–281.

[21] Stuart J. Russell and Peter Norvig. 2021. *Artificial Intelligence: A Modern Approach–Chapter 3* (4 ed.). Pearson.

[22] Katie Salen and Eric Zimmerman. 2003. *Rules of Play: Game Design Fundamentals*. The MIT Press, Cambridge, MA, USA.

[23] MK Saravana, G Manjula, Nayan Shekar, Pasyam Harshavardhan Reddy, Shreecharan Dattatreya Hegde, and BN Koushik. 2025. Cognitively-Grounded Adaptive Sudoku Generation and Real-Time Skill Assessment. In *Proceedings of the 2nd International Conference on Software, Systems and Information Technology (SSITCON)*. IEEE, 1–7. doi:10.1109/SSITCON66133.2025.11342206

[24] Noor Shaker, Julian Togelius, and Mark J. Nelson. 2016. *Procedural Content Generation in Games*. Springer International Publishing, Cham, Switzerland. doi:10.1007/978-3-319-42716-4

[25] Claude E. Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27, 3 (1948), 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x

[26] Helmut Simonis. 2005. Sudoku as a constraint problem. In *Proceedings of the 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, Vol. 12. Sitges, Spain, 13–27. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=20fbc353240e1fcd2a23eb1b9cf37060de469e32

[27] Carlos Souza, Luciana Berretta, and Sérgio Carvalho. 2025. Consolidating a MAPE-K-Based Architecture for Dynamic Difficulty Adjustment: A Bottom-Up Approach. In *Anais do I Workshop sobre Engenharia de Software para Desenvolvimento de Jogos (SE4Games)* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 49–52. doi:10.5753/se4games.2025.14865

[28] Elio Valenzuela, Hans Schaa, Nicolas A Barriga, and Gustavo Patow. 2025. Using search algorithm statistics for assessing maze and puzzle difficulty. *Entertainment Computing* 53 (2025), 100925. doi:10.1016/j.entcom.2025.100925

[29] Su Xue, Meng Wu, John Kolen, Navid Aghdaie, and Kazi A Zaman. 2017. Dynamic difficulty adjustment for maximized engagement in digital games. In *Proceedings of the 26th International Conference on World Wide Web Companion*. 465–471. doi:10.1145/3041021.3054170

[30] Georgios N. Yannakakis and Julian Togelius. 2011. Experience-Driven Procedural Content Generation. *IEEE Transactions on Affective Computing* 2, 3 (July 2011), 147–161. doi:10.1109/T-AFFC.2011.6

[31] Takayuki Yato and Takahiro Seta. 2003. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 86, 5 (2003), 1052–1060. https://search.ieice.org/bin/summary.php?id=e86-a_5_1052

[32] Jichen Zhu and Santiago Ontañón. 2020. Player-Centered AI for Automatic Game Personalization: Open Problems. In *Proceedings of the 15th International Conference on the Foundations of Digital Games* (Bugibba, Malta) *(FDG'20)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. doi:10.1145/3402942.3402951