Product Requirements Document: High-Performance Dice Notation Parser for Bun

1. Executive Summary and Strategic Context

The modern landscape of Tabletop Role-Playing Games (TTRPGs) has undergone a radical digital transformation. The shift from physical tabletops to Virtual Tabletop (VTT) environments, Discord-based play-by-post communities, and sophisticated character management applications has created a critical demand for robust computational infrastructure. At the heart of every digital RPG interaction lies the dice engine—a component responsible for translating human-readable stochastic instructions into deterministic, mathematically rigorous outcomes.

While the JavaScript ecosystem offers several dice parsing libraries, the current market is fragmented and plagued by legacy technical debt. Existing solutions often suffer from bloated dependency chains, reliance on slow and fragile Regular Expression engines, or a lack of support for the nuanced mechanics found in complex systems like Pathfinder 2nd Edition (PF2e) and World of Darkness (WoD). Furthermore, many libraries fail to address critical edge cases, such as the correct handling of negative numbers or the complex operator precedence required by nested mathematical expressions.

This Product Requirements Document (PRD) outlines the comprehensive specifications for a next-generation, high-performance dice parsing library. Designed specifically for the Bun runtime environment and written in strict TypeScript, this library aims to set a new industry standard for speed, type safety, and feature completeness. The architectural strategy rejects the limitations of Recursive Descent and PEG parsers in favor of a Pratt Parser (Top-Down Operator Precedence) approach. This choice is strategic, addressing the specific linguistic ambiguities of dice notation while ensuring the extensibility required for future game systems.

The proposed implementation is structured into three progressive stages. Stage 1 establishes the Core Engine and D&D 5e compatibility, focusing on the resolution of foundational architectural challenges. Stage 2 expands into System Compatibility, introducing the complex modifier logic required for Pathfinder and dice-pool systems. Stage 3 delivers Advanced Features, including macro expansion, variable injection, and semantic output formatting. This phased approach ensures immediate utility while systematically eliminating the technical deficits present in current open-source offerings.

2. Theoretical Foundations and Architectural Analysis

To engineer a parser that surpasses existing solutions, one must first deconstruct the linguistic and mathematical properties of dice notation. It is insufficient to treat dice rolls as simple text replacement tasks; rather, standard dice notation (NdX) functions as a

Domain-Specific Language (DSL) with high operator density and context-sensitive grammar.

2.1 The Linguistic Complexity of Dice Notation

Superficially, dice notation appears to be simple arithmetic. However, it combines algebraic operations, stochastic functions (the rolls themselves), and set operations (keep/drop/sort) into a single, compact syntax. This creates unique parsing challenges that baffle standard mathematical parsers.

The primary source of ambiguity is the d operator. In the expression 4d6, d acts as a binary (infix) operator, taking a count on the left and a size on the right. However, in the expression d20, d acts as a unary (prefix) operator, implying a count of one. Furthermore, in the context of modifiers, d can also signify "drop," as in 4d6d1 (roll 4 six-sided dice, drop the lowest 1). A robust parser must disambiguate these contexts without relying on fragile lookahead regexes that degrade performance and maintainability.

A defining failure of current libraries is the "Negative Number Bug." Standard arithmetic dictates that 3 - 5 equals -2. However, in a dice context, users often write expressions like 1d4-1. If the die rolls a 1, the result is 0. If the system is used for a penalty calculation, users might write 3d4 - 7. If the roll is 4, the result must be -3. Many existing parsers, specifically those relying on basic regex substitution or improperly configured operator precedence, incorrectly clamp these results to 0 or 1, or fail to parse the negative sign as a unary negation operator versus a binary subtraction operator.[1]

2.2 Architectural Selection: The Superiority of Pratt Parsing

The analysis of parsing methodologies reveals three primary candidates: Recursive Descent, Parsing Expression Grammars (PEG), and Pratt Parsing (Top-Down Operator Precedence).

Recursive Descent is the traditional choice for many hand-written parsers. It maps grammar rules directly to functions (e.g., parseExpression, parseTerm). While intuitive, Recursive Descent struggles with left-recursive grammars and operator precedence without verbose grammar definitions. As the dice DSL grows to include dozens of modifiers (!, !!, !p, r, ro, kh, kl), the function call depth and complexity of a Recursive Descent parser become unmanageable.[1]

PEG (Parsing Expression Grammar) parsers, often generated by tools like PEG.js, offer a declarative syntax. However, PEGs often suffer from performance penalties due to memoization overhead (Packrat parsing) and difficulties handling left recursion naturally. For a library targeting high-frequency usage in serverless environments (Bun), the cold-start time and memory footprint of generated PEG parsers are suboptimal.[2]

Pratt Parsing emerges as the optimal architecture for dice notation. Developed by Vaughan Pratt, this technique associates parsing logic directly with tokens rather than grammar rules. Each token is assigned a "binding power" (precedence) and up to two semantic functions:

- **Null Denotation (NUD):** Handles tokens appearing at the start of an expression (literals, prefix operators).
- **Left Denotation (LED):** Handles tokens appearing between expressions (infix operators, postfix modifiers).

This architecture solves the core domain challenges elegantly:

1. **Contextual Behavior:** The d token can have both a NUD (for d20) and an LED (for 4d6), resolving the unary/binary conflict without special grammar rules.
2. **Dynamic Precedence:** Pratt parsing handles the mix of left-associative operators (subtraction) and right-associative operators (exponentiation **) natively via binding power comparison.[4]
3. **Extensibility:** Adding a new modifier, such as "Penetrating Dice" (!p), involves registering a single new token handler. It does not require refactoring a complex grammar tree, making the library highly maintainable.[2]

2.3 The Bun Runtime Advantage

The choice of Bun as the target runtime is strategic. Bun's JavaScriptCore engine is optimized for rapid startup and execution, characteristics critical for serverless functions (e.g., AWS Lambda or Cloudflare Workers) that often power Discord bots. A dice parser in this environment may be cold-started thousands of times to process individual commands. Bun's native support for TypeScript eliminates the need for build-step transpilation during development, and its built-in test runner (bun test) provides a high-performance harness for the thousands of property-based tests required to verify stochastic correctness.[5]

3. Stage 1: Core Engine and D&D 5e Foundations

The initial stage focuses on building the parsing infrastructure and supporting the core mechanics of Dungeons & Dragons 5th Edition. This represents the Minimum Viable Product (MVP) necessary to satisfy the largest segment of the user base.

3.1 Lexical Analysis (Tokenizer) Specifications

The Lexer's responsibility is to transform raw input strings into a stream of typed tokens. It must be resilient to user error, variable whitespace, and case insensitivity—typical characteristics of manual input in chat applications.
Requirement 1.1: Token Taxonomy
The library must define a strict, comprehensive set of tokens. In TypeScript, these should be defined via enum or union types to enforce strict checking throughout the parsing pipeline.

- **NUMBER:** Matches integers and floating-point numbers. Pattern: /[0-9]+(\.[0-9]+)?/. Used for dice counts, face counts, and modifier values.
- **DICE:** Matches d or D. This token identifies a dice roll instruction.
- **PLUS / MINUS:** Matches + and -. Used for arithmetic and unary negation.
- **MULTI / DIV / MOD / POW:** Matches *, /, %, ** (or ^). Essential for calculating damage resistance (halving) or crits (multiplication).
- **LPAREN / RPAREN:** Matches (and). Critical for enforcing operation precedence.
- **KEEP:** Matches k, kh, kl. Used for Advantage/Disadvantage and stat generation.
- **DROP:** Matches d, dh, dl. Note the lexical collision with DICE.
- **TEXT:** Matches annotations or labels (optional at this stage but planned for output).

Requirement 1.2: Disambiguation Strategies
The Lexer must implement a "Maximal Munch" strategy to resolve the collision between the

dice operator d and the drop modifier d.

- **Logic:** When the character d is encountered, the lexer must peek ahead. If the subsequent character is h or l (forming dh or dl), it serves as a distinct DROP token. If the d is followed by a number, its semantic meaning (Dice vs. Drop) is context-dependent and should be resolved by the Parser, not the Lexer. However, distinguishing dh/dl at the token level simplifies the parser's logic significantly.[1]

3.2 Abstract Syntax Tree (AST) Design

The Parser does not calculate results directly; it builds an AST. This decoupling allows for validation, serialization, and multiple evaluations of the same formula (e.g., re-rolling a saved macro).
Requirement 1.3: AST Node Interfaces
The AST must use a Discriminated Union pattern in TypeScript to ensure type safety.

TypeScript

```
export type ASTNode =

| LiteralNode
| DiceNode
| OperatorNode
| ModifierNode
| GroupNode;

export interface LiteralNode {
  type: 'Literal';
  value: number;
}

export interface DiceNode {
  type: 'Dice';
  count: ASTNode; // Allows expressions like (1+1)d6
  sides: ASTNode; // Allows expressions like 1d(2*10)
}

export interface OperatorNode {
  type: 'Operator';
  operator: '+' | '-' | '*' | '/' | '%' | '**';
  left: ASTNode;
  right: ASTNode;
}
```

```
export interface ModifierNode {
  type: 'Modifier';
  modifier: 'keep' | 'drop';
  variant: 'highest' | 'lowest';
  count: ASTNode;
  operand: ASTNode; // The node being modified
}
```

This recursive structure is non-negotiable. It allows for "Computed Dice" (e.g., (2+2)d6), a feature often requested by power users but missing from simpler regex parsers.[1]

3.3 RNG and Determinism

A dice library is only as good as its source of entropy. Standard Math.random() is insufficient for rigorous testing and "fairness" verification.

Requirement 1.4: Seedable PRNG

The library must abstract the Random Number Generator (RNG) behind an interface.

- **Implementation:** The default implementation should use a high-quality seeded PRNG. The research recommends xorshift128+ or the random-js library (which uses Mersenne Twister or similar algorithms).[8]
- **Justification:** Seedable RNG is mandatory for unit testing. A test case must be able to inject a seed (e.g., "TEST_SEED") and assert that 1d20 results in exactly 15. This makes the library deterministic for debugging purposes.
- **Bun Integration:** Bun provides Math.random() which is generally V8/JSC compliant, but for cryptographic requirements, Bun.crypto or crypto.getRandomValues() should be accessible via configuration.[10]

3.4 Core D&D 5e Mechanics Implementation

The MVP must verify specific D&D 5e behaviors.

Requirement 1.5: Advantage and Disadvantage

- **Mechanic:** Roll two d20s, take the highest (Advantage) or lowest (Disadvantage).
- **Syntax:** 2d20kh1 (Keep Highest 1) and 2d20kl1 (Keep Lowest 1).
- **Alias Support:** Users expect shorthand. The parser should optionally map adv to 2d20kh1 and dis to 2d20kl1 during the tokenization or parsing phase.[1]

Requirement 1.6: Stat Generation

- **Mechanic:** Roll 4d6, drop the lowest die.
- **Syntax:** 4d6dl1 or 4d6kh3.
- **Verification:** The evaluator must sort the individual die results, identify the lowest value(s), and exclude them from the summation.

Requirement 1.7: Negative Number Handling

- **Mechanic:** Allow penalties to reduce a total below zero.
- **Verification:** An input of 1d4 - 5 given a roll of 1 must return -4. The library must explicitly *not* clamp results to 0 unless a max(0,...) function is explicitly invoked by the

user.[1]

3.5 Implementation Guidance: The Pratt Algorithm

For the AI or developer implementing this stage:

1. **Initialize the Parser:** Create a class Parser holding the token stream and a tokenIndex.
2. **Register NUDs:** Register NUMBER to return a LiteralNode. Register MINUS to return a unary negation operator (binding power high). Register LPAREN to handle grouped expressions.
3. **Register LEDs:** Register PLUS, MINUS, MULTI, DIV as binary operators. Register DICE (d) with a binding power higher than math but lower than parentheses.
4. **Register Modifiers:** Register KEEP (k/kh) and DROP (d/dl) as postfix operators (LEDs that do not consume a right-hand expression in the traditional infix sense, but consume parameters).
5. **Main Loop:** parseExpression(precedence) calls the NUD of the current token, then loops while the next token's binding power > precedence, calling the LED of the next token.

4. Stage 2: System Compatibility (Pathfinder & WoD)

Stage 2 expands the library to support the complex ecosystems of Pathfinder (1e/2e) and World of Darkness. These systems introduce non-standard arithmetic, complex recursion, and result counting.

4.1 Advanced Modifiers: Exploding and Rerolling

These modifiers alter the process of rolling itself, potentially generating infinite sets of numbers.

Requirement 2.1: Exploding Dice (!)

- **Mechanic:** If a die rolls the maximum value (or a target $>N$), roll another die and add it to the set.
- **Syntax:** NdX! (explode on max) or NdX!>Y (explode on range).
- **Compounding (!!):** Shadowrun style. The extra rolls are added to the *value* of the original die, not the *count* of dice. 1d6!! rolling a 6 then a 5 results in a single die value of 11, not two dice (6, 5).[1]
- **Penetrating (!p):** HackMaster style. Exploded dice subtract 1 from their result. 1d6!p rolling 6, then 6, then 2 results in $6 + (6-1) + (2-1) = 12$.
- **Safety:** The engine must enforce a hard recursion limit (e.g., 1000 iterations) to prevent infinite loops (e.g., 1d1!) from crashing the Bun process.[1]

Requirement 2.2: Reroll Mechanics (r, ro)

- **Reroll (r):** Reroll *any* die meeting the condition, recursively. 1d6r<2 rerolls 1s until a number ≥ 2 appears.
- **Reroll Once (ro):** Crucial for D&D 5e "Great Weapon Fighting". Reroll matching dice exactly once; accept the second result even if it matches the condition.
- **Verification:** 2d6ro<3 must allow a sequence of [2 -> 1] to result in 1. Standard r would

continue rerolling the 1.

4.2 Pathfinder 2e Mechanics: Degrees of Success

PF2e utilizes a four-tier success system that cannot be represented by a single boolean or number.

Requirement 2.3: Degree of Success Logic

- **The Math:** A check result is compared to a DC.
    - **Critical Success:** Result $\ge DC+10$.
    - **Success:** $DC \le Result < DC+10$.
    - **Failure:** $DC-10 < Result < DC$.
    - **Critical Failure:** Result $\le DC-10$.
- **The "Nat 20/1" Rule:** A Natural 20 upgrades the degree by one step. A Natural 1 downgrades it by one step. It is *incorrect* to treat a Nat 20 as an automatic critical success; it is merely a step-up.[11]
- **Output Requirement:** The parser must return the total and the raw die_value separately. It should ideally allow a comparison syntax like 1d20+10 vs 25 which returns a structured object indicating the degree of success (0-3).[13]

4.3 Success Counting (Dice Pools)

Systems like WoD or Shadowrun do not sum dice. They count how many dice exceed a threshold.

Requirement 2.4: Target Numbers (>)

- **Syntax:** NdX>Y. Example: 10d10>6 (World of Darkness).
- **Logic:** The evaluator iterates through the rolled array. For each die d, if $d \ge Y$, increment success count.
- **Failure Counting (f):** Example: 10d10>6f1. If $d \ge 6$, success +1. If $d = 1$, success -1.
- **Integration:** In the Pratt parser, > acts as an infix operator. However, unlike math operators, it transforms a DiceResult (array) into a CountResult (integer). This requires the AST evaluation phase to handle heterogeneous return types.[1]

4.4 Grouped Rolls and Set Operations

Grouped rolls allow users to perform operations on multiple distinct expressions.

Requirement 2.5: Group Syntax {}

- **Syntax:** { 1d8+4, 1d10+2 }.
- **Functionality:**
    - **Keep/Drop:** {...}kh1 executes both sub-expressions and keeps the one with the higher total.
    - **Sum:** {...} creates a set. sum({...}) adds them.
- **Implementation:** The lexer detects {. The parser treats the contents as a comma-separated list of expressions, returning a GroupNode. The evaluator resolves all children before applying modifiers.[14]

4.5 Math Functions

Pathfinder macros rely heavily on rounding functions.
Requirement 2.6: Supported Functions

- **List:** floor(), ceil(), round(), abs(), max(), min().
- **Pathfinder Nuance:** Integer division in Pathfinder usually rounds down. The library should support floor(A/B) syntax.
- **Parsing:** These are treated as prefix operators or function calls within the Pratt parser, binding tightly to their parenthesized arguments.[1]

5. Stage 3: Advanced Features and API Parity

The final stage aims for feature parity with established VTTs like Roll20 and Foundry, creating a developer-centric API that supports macros and rich output.

5.1 Macro and Variable Expansion

Rolls often depend on character statistics not known at parse time.
Requirement 3.1: Context Injection

- **Feature:** The roll function must accept a context object (dictionary of key-value pairs).
- **Syntax:** 1d20 + @strMod or 1d20 + @{strength}.
- **Process:** The lexer identifies @. The parser creates a VariableNode. During evaluation, the engine looks up the variable name in the context. If missing, it should throw a descriptive error or default to 0 (configurable).[1]

5.2 Structured JSON Output

For a VTT, the final number is less important than the "story" of the roll. Users need to see *which* dice were dropped, which exploded, and which were criticals.
Requirement 3.2: Output Schema
The library must return a rich object, not just a number.

TypeScript

```
interface RollResult {
  total: number;
  notation: string;
  rendered: string; // "1d20(15) + 5 = 20"
  parts: Array<{
    type: 'die' | 'operator' | 'literal';
    value: number;
    rolls?: Array<{
      result: number;
      sides: number;
      modifiers: string; // ["dropped", "exploded"]
      critical: boolean; // Natural 20 or max value
      fumble: boolean;   // Natural 1 or min value
    }>;
  }>;
}
```

This structure enables frontend clients (React/Vue/Svelte) to render 3D dice or visual logs without re-parsing the string.[15]

5.3 Advanced Sorting and Visualization

- **Sorting (s):** 4d6s does not change the mathematical total but reorders the rolls array in the JSON output (ascending). sd sorts descending.
- **Critical/Fumble Thresholds (cs, cf):** 1d20cs>19 marks 19 and 20 as criticals in the output metadata. This allows systems like Champion Fighter (crit on 19) to be represented visually without altering the sum logic.[16]

6. Implementation Guidelines for AI Agents

When generating code for this library, AI agents should adhere to the following strictures to ensure quality and alignment with the PRD.

6.1 Code Style and Structure

- **Language:** Strict TypeScript. No any. Use unknown with type guards where necessary.
- **Bundling:** Configure bun build to emit both ESM (.mjs) and CJS (.js) formats to ensure broad compatibility, although the primary target is Bun.[17]
- **Testing:** Use bun test. Tests must be co-located with source files (e.g., parser.ts and parser.test.ts).

6.2 The "Negative Number" Verification

Every AI-generated parser iteration must pass this specific regression test:

TypeScript

```typescript
test("Parses negative numbers correctly", () => {
  // Mock RNG to return 2
  const result = roll("1d4 - 5", { seed: "force_2" });
  expect(result.total).toBe(-3);
});
```

This ensures the parser correctly identifies binary subtraction versus unary negation.[1]

6.3 Performance Optimization

- **Object Allocation:** Minimize object creation in the roll hot path. Re-use token objects if possible.
- **Lookup Tables:** Use Map or pre-computed lookup tables for Operator Precedence (Binding Power) rather than switch statements, as Bun's JIT optimizes these efficiently.

6.4 Verification via Property-Based Testing

Unit tests are insufficient for random behavior. Use fast-check (compatible with Bun) to define properties.

- **Invariant:** min_possible <= result <= max_possible.
- **Invariant:** result is always an integer (unless math functions introduce floats).
- **Invariant:** The length of the rolls array in the output object matches the number of dice requested (accounting for explosions).[18]

7. Roadmap and Conclusion

This architecture provides a clear path to a market-leading dice library. By leveraging Pratt Parsing, we solve the historical fragility of dice syntax. By targeting Bun, we achieve the performance necessary for high-scale bot infrastructure.

- **Phase 1:** Core Lexer, Pratt Parser, Basic Arithmetic, Keep/Drop. (Target: D&D 5e MVP).
- **Phase 2:** Exploding, Rerolling, Success Counting, Seeded RNG. (Target: Pathfinder/WoD).
- **Phase 3:** Grouped Rolls, Variables, Math Functions, Rich JSON Output. (Target: Roll20 Parity).

This document serves as the single source of truth for the development lifecycle, ensuring that every line of code contributes to a robust, extensible, and high-performance engine.
Data Sources Cited: [1]

Works cited

1. Deep Research - Roll Robot - Gemini.pdf
2. Pratt Parsers: Expression Parsing Made Easy – journal.stuffwithstuff.com, accessed December 27, 2025, https://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/
3. Difference between a PEG and recursive descent parser? - Stack Overflow, accessed December 27, 2025, https://stackoverflow.com/questions/59157302/difference-between-a-peg-and-recursive-descent-parser
4. How do Pratt parsers compare to other parser types and why are they used so little?, accessed December 27, 2025, https://stackoverflow.com/questions/13634393/how-do-pratt-parsers-compare-to-other-parser-types-and-why-are-they-used-so-litt

5. Benchmarking - Bun, accessed December 27, 2025, https://bun.com/docs/project/benchmarking
6. Bun — A fast all-in-one JavaScript runtime, accessed December 27, 2025, https://bun.com/
7. Bun vs Node.js 2025: Performance, Speed & Developer Guide - Strapi, accessed December 27, 2025, https://strapi.io/blog/bun-vs-nodejs-performance-comparison-guide
8. Don't use Math.random() - DeepSource, accessed December 27, 2025, https://deepsource.com/blog/dont-use-math-random
9. LinusU/node-xorshift128plus: xorshift128+ implementation for Node.js - GitHub, accessed December 27, 2025, https://github.com/LinusU/node-xorshift128plus
10. Seeding the random number generator in JavaScript - Stack Overflow, accessed December 27, 2025, https://stackoverflow.com/questions/521295/seeding-the-random-number-generator-in-javascript
11. accessed December 27, 2025, https://www.reddit.com/r/Pathfinder2e/comments/1g0h8nt/effects_that_decrease_or_increase_degrees_of/#:~:text=This%20means%20that%20a%20natural,or%20more%20below%20the%20DC.
12. How do criticals interact with degrees of success for attacks? - RPG Stack Exchange, accessed December 27, 2025, https://rpg.stackexchange.com/questions/175240/how-do-criticals-interact-with-degrees-of-success-for-attacks
13. Step 4: Determine the Degree of Success and Effect - Pathfinder 2 - pf2easy.com, accessed December 27, 2025, https://pf2easy.com/index.php?id=5753&name=Step_4:_Determine_the_Degree_of_Success_and_Effect
14. Dice Reference - Roll20 Help Center, accessed December 27, 2025, https://help.roll20.net/hc/en-us/articles/360037773133-Dice-Reference
15. API: Chat - Roll20 Help Center, accessed December 27, 2025, https://help.roll20.net/hc/en-us/articles/360037256754-API-Chat
16. [Roll20] How to roll dice like a MOTHERFUCKER [Surprisingly SFW] : r/DnD - Reddit, accessed December 27, 2025, https://www.reddit.com/r/DnD/comments/5ejl1l/roll20_how_to_roll_dice_like_a_motherfucker/
17. Bundler - Bun, accessed December 27, 2025, https://bun.com/docs/bundler
18. How to get started with property-based testing in JavaScript using fast-check, accessed December 27, 2025, https://jrsinclair.com/articles/2021/how-to-get-started-with-property-based-testing-in-javascript-with-fast-check/
19. Exploring RPG Dice Systems: A Comprehensive Guide - Rune Rollers, accessed December 27, 2025, https://runerollers.com/blogs/dnd-dice/exploring-rpg-dice-systems-a-comprehensive-guide