Reflection-based JSON in C++ at Gigabytes per Second

CppCon 2025

Daniel Lemire & Francisco Giordano Thiesen

daniel@lemire.me, *franciscogthiesen@gmail.com*

What This Talk Is About

The Promise: Automated JSON ↔️ C++ struct conversion at gigabyte speeds

The Reality: We built it using experimental C++26 reflection

The Catch: Still requires Bloomberg's LLVM fork

The Future: Production-ready when compilers catch up

Why JSON Matters

```json
{
  "age": 5,
  "name": "Daniel",
  "toys": ["wooden dog", "little car"]
}
```

- **Ubiquitous**: Web APIs, config files, data interchange
- **Human-readable**: Intuitive attribute-value pairs and arrays
- **But**: Performance bottleneck + tedious manual work in C++

The C++ JSON Pain

Performance Bottlenecks:

- Complex string handling
- Nested structure parsing
- Manual struct ↔️ JSON mapping overhead
- Number parsing and serialization costs

Safety Pitfalls:

- Runtime type mismatches
- Unexpected input (broken Unicode)
- No compile-time guarantees

Other Languages Have It Easy

Rust (serde):

```rust
#[derive(Serialize, Deserialize)]
struct Person {
    age: u32,
    name: String,
}
// That's it! Automatic JSON conversion
```

Similar ergonomics in: Python, Go, C#, Java, Zig

C++: Still manual work... until now

C++20 Sets the Stage

Concepts: Cleaner, constrained interfaces

```
template<Serializable T>
void to_json(const T& obj);
```

tag_invoke: Customization point mechanism

constexpr: Compile-time computation

These lay groundwork but fall short of full automation

Enter C++26 Reflection (PR2996)

What it is: Introspect C++ types at compile time

- Field names, types, structure layout
- "The compiler knows your data structures"

Why it's game-changing: No more manual mapping!

Current state: Bloomberg's experimental LLVM fork

- github.com/bloomberg/clang-p2996

Simple Reflection Example

```cpp
struct Person {
    int age;
    std::string name;
};

// Reflection automatically exposes:
// - Field count: 2
// - Field names: "age", "name"
// - Field types: int, std::string
// - Automatic JSON mapping possible!
```

Our Implementation Strategy

Core Idea: Reflection auto-generates JSON mapping code

Serialization: `Person` → `{"age": 5, "name": "Daniel"}`

- Walk struct fields via reflection
- Emit JSON strings (compile-time optimized)

Deserialization: JSON → `Person`

- Parse tokens with type safety
- Populate struct fields automatically
- Comprehensive error handling

Implementation Architecture

Three-layer design:

1. **Lexer**: Tokenize JSON input
2. **Parser**: Build structured representation
3. **Reflection Binder**: Map to/from C++ structs

Key insight: Reflection happens at compile time

- Field names become string literals
- Type safety enforced by compiler
- Zero runtime reflection overhead

First Prototype: Reality Check

```cpp
// Our first attempt was... not impressive
struct Data { /* ... */ };
auto start = std::chrono::high_resolution_clock::now();
Data result = parse_json<Data>(json_string);
auto end = std::chrono::high_resolution_clock::now();
// Result: Much slower than expected!
```

Problems: Naive string handling, reflection overhead

Lesson: Measure first, optimize second

Optimization Journey

Key Performance Wins:

1. **Compile-time key preparation**

```
constexpr auto field_names = get_field_names<T>();
// Keys computed at compile time!
```

2. **SIMD string operations**
   - Leveraged existing simdjson SIMD expertise
3. **Memory layout optimization**
   - Better cache locality for reflection data

The SIMD Advantage

```
// Traditional approach
for (char c : json_string) {
    if (c == '"') { /* handle string */ }
    // Character-by-character processing
}

// SIMD approach (simplified)
__m256i chunk = _mm256_loadu_si256(ptr);
__m256i quotes = _mm256_cmpeq_epi8(chunk, quote_chars);
// Process 32 characters at once!
```

Building on simdjson's proven SIMD foundations

Performance Result: The Leap

Before optimization: Hundreds of MB/s

After optimization: **Gigabytes per second**

Exact numbers from our ablation study coming...

Note: The profiling experience was… educational

- Some bottlenecks were surprising
- SIMD made the biggest difference
- Compile-time optimization crucial

Serialization Benchmarks

Test Setup

- **Hardware**: [To be filled with actual specs]
- **Test data**: Complex nested JSON structures
- **Competitors**: nlohmann/json, RapidJSON, others

Results Preview

```
Our Implementation:    X.X GB/s
nlohmann/json:        Y.Y GB/s
RapidJSON:            Z.Z GB/s
```

It's really fast! (But could be faster)

Deserialization: Head-to-Head

The Competition

- **Rust serde**: The gold standard
- **nlohmann/json**: Popular C++ choice
- **RapidJSON**: Performance-focused C++
- **simdjson**: Our non-reflection baseline

Our Advantage

Reflection + C++ control + SIMD = Winning combination

Detailed numbers and charts coming in ablation study

System Dependencies Matter

Performance varies by processor:

- AVX2 vs AVX-512 differences
- Memory bandwidth impact
- Cache architecture effects

Lesson: Always benchmark on target hardware

Good news: Fast across different architectures

Safety: What Did We Gain?

Compile-time Benefits:

```cpp
struct Person { int age; std::string name; };

// Type mismatch caught at compile time:
Person p = parse_json<Person>(R"({"age": "not_a_number"})");
// ↑ Compiler error, not runtime surprise!
```

Runtime Benefits:

- Graceful error handling for malformed JSON
- Clear diagnostic messages
- Structured error reporting

Safety Trade-offs

Better than handwritten:

- No manual field mapping bugs
- Type safety enforced automatically
- Consistent error handling

Not foolproof:

- Runtime edge cases still possible
- JSON schema evolution challenges
- Unicode edge cases remain tricky

Net result: Significant safety improvement

Downside #1: Learning Curve

Reflection is new territory:

```cpp
template<typename T>
constexpr auto serialize(const T& obj) {
    constexpr auto members = std::reflect::members_of(^T);
    // What does ^T mean? How does this work?
    return [=]<std::size_t... Is>(std::index_sequence<Is...>) {
        return make_json_object(
            get_name(members[Is])...,
            obj.[:members[Is]:]...
        );
    }(std::make_index_sequence<members.size()>{});
}
```

"Template metaprogramming's wild cousin—powerful but untamed"

Downside #2: Tooling Challenges

Debugging is tricky:

- Reflection errors can be cryptic
- IDE support still catching up
- Template instantiation debugging nightmares

Example debugging session:

```
error: no viable conversion from 'std::reflect::info' to 'const char*'
  in instantiation of 'serialize<Person>' requested here
    note: in instantiation of 'get_field_name<0>' requested here
      note: candidate template ignored: substitution failure
```

Sound familiar, template veterans?

Downside #3: Build Times

The reality check:

```
# Without reflection
$ time make
real    0m15.234s

# With reflection
$ time make
real    1m23.891s
```

Why slower:

- Reflection generates substantial code
- LLVM fork not optimized for compilation speed
- Template instantiation explosion

Build Time Mitigation

Strategies we explored:

1. **Precompiled headers**

   ```
   // precompiled_reflection.h
   #include <reflect>
   #include "common_types.h"
   ```

2. **Selective compilation**
   - Only reflection-enable critical paths
   - Traditional parsing for less critical code
3. **Caching strategies**
   - ccache with reflection-aware hashing

Downside #4: Error Messages

The brutal truth:

```
error: invalid use of incomplete type 'std::reflect::member_info<
  std::reflect::get_public_data_members_t<Person>[0]>'
  in instantiation of function template specialization
  'get_member_name<Person, 0>' requested here
    note: in instantiation of function template specialization
    'serialize_impl<Person>' requested here
      note: while substituting template arguments for class template
```

Real errors from our development

Hope: Compiler vendors will improve diagnostics

When Should You Try This?

Today: Experimental only

- Bloomberg LLVM fork required
- Limited ecosystem support
- Expect rough edges

Near future (1-2 years): Early adopters

- GCC/Clang support incoming
- Better tooling
- Production experiments

Future (3+ years): Mainstream ready

- Full compiler support

How Beneficial Is It?

Developer productivity:

- Eliminate boilerplate JSON mapping
- Automatic type safety
- Consistent error handling

Performance gains:

- Gigabyte-per-second parsing
- Compile-time optimizations
- SIMD acceleration built-in

Maintenance benefits:

- Schema changes automatically handled
- Single source of truth (struct definitions)

How Impactful Will This Be?

For C++ ecosystem:

- Closes ergonomics gap with other languages
- Enables rapid prototyping with high performance
- Makes C++ more attractive for data-intensive applications

For your projects:

- Faster development cycles
- Better performance than hand-written code
- Safer, more maintainable JSON handling

Game-changer for data-driven C++

Code Examples: Before and After

Traditional C++ approach:

```cpp
struct Person {
    int age;
    std::string name;
    std::vector<std::string> toys;
};

// Manual JSON handling (20+ lines of boilerplate)
void to_json(json& j, const Person& p) {
    j = json{{"age", p.age}, {"name", p.name}, {"toys", p.toys}};
}
void from_json(const json& j, Person& p) {
    j.at("age").get_to(p.age);
    j.at("name").get_to(p.name);
    j.at("toys").get_to(p.toys);
}
```

Code Examples: With Reflection

Reflection-based approach:

```
struct Person {
    int age;
    std::string name;
    std::vector<std::string> toys;
};

// That's it! Everything else is automatic:
std::string json_str = R"({"age": 5, "name": "Daniel", "toys": ["car"]})";
Person p = simdjson::parse<Person>(json_str);
std::string output = simdjson::serialize(p);
```

Zero boilerplate. Maximum performance. Compile-time safety.

Advanced Features

Customization hooks:

```
struct Config {
    int timeout_ms;
    std::string endpoint;

    // Custom field naming
    static constexpr auto json_field_names() {
        return std::make_tuple("timeout", "api_endpoint");
    }
};
```

Validation integration:

```
struct ValidatedData {
    int count;

    void validate() const {
        if (count < 0) throw std::invalid_argument("count must be non-negative");
```

Future Roadmap

Short term (2025):

- Benchmark suite completion
- Error message improvements
- More comprehensive validation

Medium term (2026-2027):

- Mainstream compiler support
- Ecosystem integration (fmt, ranges, etc.)
- Production hardening

Long term (2028+):

- Standard library integration

Related Work & Acknowledgments

Building on giants:

- simdjson SIMD foundations
- Bloomberg's reflection implementation
- Rust serde design inspiration
- C++20 concepts and customization points

Community efforts:

- P2996 reflection proposal authors
- EWG feedback and guidance
- Early adopters and testers

Practical Considerations

Memory usage:

- Reflection metadata is compile-time only
- Runtime overhead minimal
- SIMD requires aligned allocations

Thread safety:

- Parser instances are thread-local
- Reflection data is read-only
- Concurrent parsing fully supported

Integration:

- Header-only design (when possible)

Questions for the Audience

For library authors:

- How would you integrate this into existing JSON workflows?
- What customization points are most important?

For application developers:

- What's your biggest JSON performance bottleneck?
- How much boilerplate would this eliminate for you?

For the curious:

- What other domains could benefit from reflection?

Live Demo

What we'll show:

1. Simple struct → JSON conversion
2. Complex nested data parsing
3. Performance comparison
4. Error handling in action

[Demo section - actual code execution]

Note: Running on Bloomberg LLVM fork

- Reflection syntax may evolve
- Performance representative of potential

Takeaways

Key insights:

1. Reflection + Performance = Possible in C++
2. SIMD makes dramatic difference
3. Compile-time optimization crucial
4. Safety improvements significant
5. Tooling challenges remain

Bottom line: The future of C++ JSON handling looks bright

Call to action: Try it on Bloomberg's fork, give feedback!

Resources

Code & Documentation:

- Implementation: github.com/simdjson/simdjson/tree/reflection_based_serialization
- Bloomberg LLVM: github.com/bloomberg/clang-p2996
- Reflection proposal: P2996

Contact:

- daniel@lemire.me
- franciscogthiesen@gmail.com

All source code available under business-friendly license

Questions?

Thank you for your attention!

Reflection-based JSON in C++ at Gigabytes per Second

Ready for the data-driven age

Bonus: Implementation Deep Dive

For the curious - detailed technical discussion

Reflection introspection:

```
template<typename T>
constexpr auto get_field_info() {
    constexpr auto members = std::reflect::members_of(^T);
    return []<std::size_t... Is>(std::index_sequence<Is...>) {
        return std::make_tuple(
            std::reflect::name_of(members[Is])...
        );
    }(std::make_index_sequence<members.size()>{});
}
```

Bonus: SIMD Integration Details

String parsing with AVX2:

```
__m256i load_and_validate_utf8(__m256i chunk) {
    // Validate UTF-8 sequences
    __m256i high_nibbles = _mm256_and_si256(
        _mm256_srli_epi32(chunk, 4), _mm256_set1_epi8(0x0F));

    // Character classification for JSON
    __m256i whitespace = _mm256_cmpeq_epi8(chunk, _mm256_set1_epi8(' '));
    __m256i quotes = _mm256_cmpeq_epi8(chunk, _mm256_set1_epi8('"'));

    return _mm256_or_si256(whitespace, quotes);
}
```

This is where the gigabytes per second come from