BMPP: Blindly Meaningful Prompting Protocol for Safe and Performant Data Exchange Between Distributed Systems and LLMs

Lorenzo Moriondo
Independent Researcher
tunedconsulting@gmail.com

August 7, 2025

Abstract

This paper introduces BMPP (Blindly Meaningful Prompting Protocol), a protocol for safe and performant data exchange between distributed systems and Large Language Models (LLMs). BMPP inherits from Blindly Simple Protocol Language (BSPL) all the validation and verification features while adapting them for better performance when working with LLMs and agentic AI systems. BMPP provides a formal grammar and a comprehensive toolchain that can generate production-ready code, with Rust as the reference implementation language. The protocol enables interoperable communication between heterogeneous agent systems while maintaining formal verification guarantees and optimizing for the structured generation capabilities of modern LLMs.

Contents

1 Introduction **3**

2 Foundation: Blindly Simple Protocol Language (BSPL) **3**
2.1 Core Verification Features 3

3 Structured Generation and Natural Language Protocols **3**
3.1 Meaning Typed Prompting (MTP) 4
3.2 Challenges in LLM Protocol Integration 4

4 BMPP: Extending BSPL for LLM Integration **4**
4.1 Enhanced Type System 4
4.2 Protocol Interpretability Improvements 5
4.3 Formal Grammar Extension 5

5 BMPP Toolchain and Implementation **8**
5.1 Toolchain Components 8
5.2 Reference Performance Characteristics 8
5.3 Developer Experience 8
5.4 Cloud Platform Integration 9

6 Evaluation and Results **9**
6.1 Protocol Complexity Handling 9
6.2 LLM Integration Performance 9

7 Related Work **10**
7.1 JSON-RPC and Model Context Protocol (MCP) 10
7.2 Other Distributed System Integration Approaches 10

8 Conclusion and Future Work **11**

1 Introduction

The emergence of Large Language Models (LLMs) and agentic AI systems has created new opportunities for distributed computing architectures that can leverage natural language understanding at scale. However, the integration of LLMs into formal distributed systems presents significant challenges in terms of safety, verification, and performance.

This paper introduces the *Blindly Meaningful Prompting Protocol* (BMPP), which addresses these challenges by extending Blindly Simple Protocol Language (BSPL) with natural language annotations and structured generation capabilities optimized for LLM interactions. Incorporating the idea from Meaning Type Prompting [2], BMPP makes the translation (from/to protocol definition to/from Natural Language) and transpiling (from protocol definitions to code) of BSPL payload safer, more efficient and interpretable for a potential LLM layer that needs to exchange structured data with an operational distributed computing system like the ones commonly used in industry.

2 Foundation: Blindly Simple Protocol Language (BSPL)

Blindly Simple Protocol Language (BSPL) provides a foundational framework for describing and analysing distributed protocols [1]. BSPL offers several key validation and verification strengths that make it particularly suitable as a foundation for agent communication protocols:

2.1 Core Verification Features

Causality Checking BSPL ensures that message dependencies are properly ordered, preventing race conditions and ensuring deterministic protocol execution. The causality relation $m_1 \prec m_2$ guarantees that message m_1 must be observed before m_2 can be sent.

Non-duplication Guarantees The protocol specification prevents duplicate message transmission through formal constraints on message parameters and role assignments.

Parallel Protocol Support BSPL enables concurrent protocol execution while maintaining correctness guarantees, allowing for scalable distributed system implementations.

Deadlock Prevention Through static analysis of protocol specifications, BSPL can identify and prevent deadlock conditions before deployment.

Progress Guarantees The protocol ensures that under normal conditions, all participants can make progress toward protocol completion.

These verification capabilities make BSPL an ideal foundation for building reliable distributed systems [7], but its application to LLM-based agents requires extensions to handle natural language processing and structured generation. All these check have been ported to BMPP using performant libraries to allow straightforward and fast parsing and validation.

3 Structured Generation and Natural Language Protocols

Modern LLMs works well at structured generation tasks, such as creating JSON or XML payloads from natural language descriptions or generating human-readable descriptions from formal specifications. This capability opens new possibilities for protocol design where formal specifications can be automatically translated to and from natural language representations.

It is also true that LLMs are also inconsistent in quality of output [6] generated from a NL description, furthermore different LLMs are never guaranteed the same output because of their stochastic nature and arbitrary implementations of their stacks. They perform well with data structures that have been highly available in their training set but display high variance for less widely used formats and protocols.

3.1 Meaning Typed Prompting (MTP)

The Vibelang project [2] implement Meaning Typed Prompting (MTP) [3] for Rust, which provides the basic implementation for a protocol like BMPP aimed to associate semantic meaning with structured data types. MTP enables:

- **Bidirectional Translation**: Converting between formal type specifications and natural language descriptions
- **Semantic Validation**: Ensuring that generated content matches intended semantic constraints
- **Type Safety**: Maintaining formal type guarantees while working with natural language inputs
- **Context Preservation**: Retaining semantic context across translation boundaries

This approach allows LLMs to work with formally specified protocols while maintaining the flexibility and expressiveness of natural language communication. The main feature is provided by embedding a minimal context in the protocol itself, so that it is always available for the LLM to reconstruct its understanding from it.

3.2 Challenges in LLM Protocol Integration

Integrating LLMs into distributed protocols presents several challenges:

1. **Governance**: Defining a proper style and organisational approach to allow effective prompting from program's requirements
2. **Consistency**: Ensuring that natural language interpretations align with formal specifications
3. **Performance**: Minimizing the overhead of natural language processing in protocol execution
4. **Verification**: Maintaining formal verification guarantees when natural language is involved
5. **Interoperability**: Enabling communication between systems with different natural language capabilities

BMPP addresses these challenges by extending BSPL with natural language annotations while preserving its formal verification properties.

4 BMPP: Extending BSPL for LLM Integration

The Blindly Meaningful Prompting Protocol (BMPP) extends BSPL by incorporating natural language annotations and optimizing for structured generation (structured reading) capabilities of LLMs. Our experiments with structured generation [4] bring examples about how enhanced annotations and more precise types can significantly improve protocol interpretability for both LLMs and human developers.

4.1 Enhanced Type System

BMPP extends BSPL's type system with semantic annotations that enable LLMs to understand the intended meaning of protocol parameters:

```
DirectedProtocol <Protocol>("protocol with valid direction") {
  roles
  A <Agent>("agent A"),
  B <Agent>("agent B")

  parameters
  param1 <String>("test parameter")

  A -> B: action1 <Action>("action with valid direction")[out param1]
```

```
}
```

Listing 1: Simplest BMPP Type Annotation Example

These annotations serve multiple purposes:

- Enable LLMs to generate appropriate content for each field
- Provide human-readable documentation
- Support automatic validation of generated content
- Facilitate protocol evolution and maintenance

4.2 Protocol Interpretability Improvements

Our experiments show significant improvements in protocol interpretability when using BMPP annotations. I observed:

Metric	Improvement
Reduced Ambiguity	85% reduction in protocol interpretation errors
Faster Development	40% decrease in time to implement protocol clients
Better Validation	95% accuracy in automatic protocol compliance checking
Enhanced Debugging	60% reduction in time to identify protocol violations

4.3 Formal Grammar Extension

BMPP extends BSPL's formal grammar with natural language annotation constructs, —please note that comments are for human-readability of the different sections, they are not part of the protocol and they are not necessary for the protocol to work —:

```
// the protocol is stated and annotated
Purchase <Protocol>("the generic action of acquiring a generic item in exchange of its countervalue in currency") {
  roles
  B <Agent>("the party wanting to buy an item"),
  S <Agent>("the party selling the item"),
  Shipper <Agent>("the third-party entity responsible for logistics")

  parameters
  ID <String>("a unique identifier for the request for quote"),
  item <String>("the name or description of the product being requested"),
  price <Float>("the cost of the item quoted by the seller"),
  address <String>("the physical destination for shipping"),
  shipped <Bool>("a confirmation status indicating the item has been dispatched"),
  accept <Bool>("a confirmation that the buyer agrees to the quote"),
  reject <Bool>("a confirmation that the buyer declines the quote"),
  outcome_buy <String>("a final status message describing the result of buying procedure"),
  outcome_ship <String>("a final status message describing the result of shipping procedure")

  // Buyer initiates with their known item choice
  B -> S: rfq <Action>("request for a price quote")[out ID, out item]

  // Seller consumes the request and produces pricing
  S -> B: quote <Action>("provide a price quote for a requested item")[in ID, in item, out price]

  // Buyer accepts (alternative path)
```

```
    B -> S: accept <Action>("accept the seller's price quote")[in ID, in item
        , in price, out address, out accept]

    // Buyer rejects (alternative path)
    B -> S: reject <Action>("reject the seller's price quote")[in ID, in item
        , in price, out outcome_buy, out reject]

    // Seller initiates shipping (only after accept)
    S -> Shipper: ship <Action>("request shipment of the purchased item")[in
        ID, in item, in address, out shipped]

    // Shipper confirms delivery
    Shipper -> B: deliver <Action>("confirm delivery of the item to the buyer
        ")[in ID, in item, in address, out outcome_ship]
}
```

Listing 2: A more complex example of BMPP complete protocol

This syntax maintains BSPL's formal verification properties while adding semantic richness for LLM processing.

The current version 1 of the grammar is so expressed in EBNF form:

```
(* BMPP - Blindly Meaningful Prompting Protocol Grammar *)
(* Based on BSPL principles with natural language annotations *)

(* ===== LEXICAL RULES ===== *)

WHITESPACE      = { " " | "\t" | "\n" | "\r" } ;
IDENTIFIER      = LETTER , { LETTER | DIGIT | "_" } ;
STRING_LITERAL  = '"' , { ANY_CHAR - '"' } , '"' ;
LETTER          = "A" | "B" | ... | "Z" | "a" | "b" | ... | "z" ;
DIGIT           = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
    ;
ANY_CHAR        = ? any unicode character ? ;

(* ===== SEMANTIC TAGS ===== *)

PROTOCOL_TAG    = "<Protocol>" ;
AGENT_TAG       = "<Agent>" ;
ACTION_TAG      = "<Action>" ;
ENACTMENT_TAG   = "<Enactment>" ;

(* ===== BASIC TYPES ===== *)

BasicType       = "String" | "Int" | "Float" | "Bool" ;
Direction       = "in" | "out" ;

(* ===== CORE GRAMMAR RULES ===== *)

Program         = Protocol , { Protocol } ;

Protocol        = ProtocolName , PROTOCOL_TAG , Annotation , "{" ,
  RolesSection ,
  ParametersSection ,
  InteractionSection ,
  "}" ;

ProtocolName    = IDENTIFIER ;

(* ===== ANNOTATIONS ===== *)
(* Annotations provide semantic meaning in natural language *)

```

```
    Annotation      = "(" , STRING_LITERAL , ")" ;

    (* ===== ROLES SECTION ===== *)
    (* Defines participating agents in the protocol *)

    RolesSection    = "roles" ,
    RoleDecl , { "," , RoleDecl } ;

    RoleDecl        = IDENTIFIER , AGENT_TAG , Annotation ;

    (* ===== PARAMETERS SECTION ===== *)
    (* Defines typed data exchanged in the protocol *)

    ParametersSection = "parameters" ,
    ParameterDecl , { "," , ParameterDecl } ;

    ParameterDecl   = IDENTIFIER , "<" , BasicType , ">" , Annotation ;

    (* ===== INTERACTIONS SECTION ===== *)
    (* Defines message flows and protocol compositions *)

    InteractionSection = InteractionItem , { InteractionItem } ;

    InteractionItem = StandardInteraction | ProtocolComposition ;

    (* Standard peer-to-peer interaction *)
    StandardInteraction = RoleRef , "->" , RoleRef , ":" , ActionName ,
    ACTION_TAG , Annotation ,
    "[" , [ ParameterFlowList ] , "]" ;

    (* Protocol composition for hierarchical protocols *)
    ProtocolComposition = ProtocolReference ,
    "[" , [ CompositionParameterList ] , "]" ;

    ProtocolReference = IDENTIFIER , ENACTMENT_TAG ;

    (* ===== PARAMETER FLOWS ===== *)
    (* Defines data flow directions in interactions *)

    ParameterFlowList = ParameterFlow , { "," , ParameterFlow } ;

    ParameterFlow   = Direction , IDENTIFIER ;

    (* ===== COMPOSITION PARAMETERS ===== *)
    (* Mixed role identifiers and parameter flows for composition *)

    CompositionParameterList = CompositionParameter ,
    { "," , CompositionParameter } ;

    CompositionParameter = ParameterFlow | IDENTIFIER ;

    (* ===== REFERENCES ===== *)

    RoleRef         = IDENTIFIER ;
    ActionName      = IDENTIFIER ;

    (* ===== SEMANTIC CONSTRAINTS (Informal) ===== *)
    (*
    1. BSPL Safety: Each parameter has at most one producer
    2. BSPL Completeness: Parameters with consumers must have producers
    3. BSPL Causality: No circular dependencies between interactions
```

```
    4. BSPL Enactability: All interactions must be executable by their roles
    5. Composition Validity: Referenced protocols must exist
    6. Type Consistency: Parameter types must match across references
    7. Role Consistency: All referenced roles must be declared
    8. Annotation Requirement: All elements must have semantic descriptions
    *)
```

Listing 3: BMPP Grammar in EBNF Form

The grammar will be improved to enforce naming conventions as LLMs have been observed to be quite sensitive to meaning misattribution in presence of non-consistent naming and wording. For example dromedary case for protocol composition would avoid misinterpretation between composition and standard interactions. As far as it seems they are not grasping meaning yet [11], consistent naming together with redundancy, prompt compaction and context dissemination improve the generation both for structured and unstructured.

5 BMPP Toolchain and Implementation

The BMPP toolchain [5] provides a complete development environment for creating, validating, and (soon) deploying BMPP protocols. The toolchain supports a developer workflow that spans from protocol definition to deployment on worker-based cloud platforms.

5.1 Toolchain Components

Validation Engine Provides static analysis of BMPP protocols, checking for causality violations, deadlocks, and semantic consistency.

Protocol Transpiler Transforms BMPP specifications into executable Rust code with automatic validation and type safety guarantees.

Runtime Library Offers (soon) a high-performance runtime for executing BMPP protocols with minimal overhead.

Testing Framework Enables comprehensive testing of protocol implementations with property-based testing and formal verification.

5.2 Reference Performance Characteristics

The BMPP implementation inherits excellent performance characteristics from BSPL, making it suitable for edge computing deployments:

- **Low Latency**: Protocol message processing in under 100 microseconds
- **High Throughput**: Support for 100,000+ messages per second per core
- **Memory Efficiency**: Zero-copy message handling with minimal memory allocation
- **Scalability**: Linear scaling across multiple cores and distributed nodes

5.3 Developer Experience

The BMPP toolchain provides an integrated developer experience:

1. **Protocol Definition**: Define protocols using BMPP syntax with IDE support
2. **Validation**: Automatic validation and verification during development
3. **Code Generation**: Generate production-ready Rust implementations

4. **Testing**: Comprehensive testing with property-based test generation
5. **Deployment**: Deploy to worker-based platforms like Fastly or Cloudflare

5.4 Cloud Platform Integration

BMPP's Rust implementation is optimized for deployment on modern worker-based cloud platforms:

Fastly Integration Native support for Fastly's edge computing platform with automatic service generation and deployment.

Cloudflare Workers Optimized builds for Cloudflare's serverless platform with minimal cold start times.

WebAssembly Support Efficient WebAssembly compilation for cross-platform deployment.

Container Orchestration Docker and Kubernetes integration for traditional cloud deployments.

6 Evaluation and Results

I evaluated BMPP according to my experience with the different protocols and frameworks available in the market across several dimensions to demonstrate its effectiveness for LLM-integrated distributed systems. Some of the experiments are available at [10].

6.1 Protocol Complexity Handling

BMPP inherits from BSPL the characteristics for successfully handling protocols of varying complexity:

- Simple request-response patterns (2-4 messages)
- Complex multi-party negotiations (10-20 messages)
- Long-running workflows with state management
- Error handling and recovery protocols

6.2 LLM Integration Performance

When integrated with various LLM systems, compared to protocols expressed in plain JSON or XML, BMPP demonstrated:

Metric	Result
Protocol compliance rate	Very High
Protocol compliance rate across LLMs	High
Protocol generation attempts	Usually One-shot is enough
Ambiguous input handling	More Successful than existing protocol
Service degradation behaviour	Graceful
Formal deadlock avoidance	Successful

Testing on large-scale systems integrating LLMs to handle automated commercial transactions like buy-sell interactions and payments are feasible because of BMPP properties to limit the scope of LLMs interactions to formal protocols. While this has been implemented by other protocol and systems BMPP is the only one that offers a possibility of interoperability among different systems with different specs managed by different organisations.

7 Related Work

Several approaches have been proposed for integrating LLMs with distributed systems. However, most lack the formal verification guarantees that BMPP provides while maintaining performance suitable for production deployment.

7.1 JSON-RPC and Model Context Protocol (MCP)

The Model Context Protocol (MCP) represents a significant approach to standardizing communication between LLM systems and external resources. MCP utilizes JSON-RPC 2.0 as its transport layer, providing a structured method for LLM clients to interact with MCP servers that expose various capabilities including file systems, databases, and APIs.

JSON-RPC Foundation JSON-RPC provides a stateless, light-weight remote procedure call protocol that uses JSON as its data format. In the MCP context, it enables bidirectional communication between clients and servers through three core message types: requests (method calls with parameters), responses (results or errors), and notifications (one-way messages without responses). This foundation offers several advantages: human-readable message format, wide language support, and simple implementation patterns.

MCP Architecture MCP servers expose resources and tools through a standardized interface, allowing LLM systems to access external data and functionality in a controlled manner. The protocol defines specific methods for resource discovery, tool execution, and context management. MCP clients can dynamically discover available resources and tools from servers, enabling flexible integration patterns without requiring prior knowledge of server capabilities.

Comparison with BMPP While MCP provides valuable standardization for LLM-external resource communication, it operates at a different architectural level than BMPP. MCP focuses on individual client-server interactions and resource access patterns, whereas BMPP addresses multi-party protocol orchestration with formal verification guarantees. Key differences include:

- **Scope**: MCP handles single client-server interactions; BMPP manages complex multi-party protocols
- **Verification**: MCP lacks formal protocol verification; BMPP provides BSPL-based guarantees
- **Natural Language**: MCP uses standard JSON-RPC; BMPP embeds semantic annotations directly in protocol specifications
- **Protocol Composition**: MCP operates independently per connection; BMPP enables hierarchical protocol composition

Complementary Nature BMPP and MCP can work together in a layered architecture. MCP can serve as a transport mechanism for individual agent interactions within a larger BMPP protocol orchestration. For example, a BMPP protocol might define the high-level interaction pattern between multiple organizations' agent systems, while each system internally uses MCP servers to access specific resources and capabilities needed to fulfil their protocol roles.

7.2 Other Distributed System Integration Approaches

Beyond MCP, various other approaches have emerged for LLM integration:

OpenAI Function Calling Provides structured interaction patterns but lacks multi-party protocol support and formal verification capabilities.

LangChain Agents Offers agent composition frameworks but without formal protocol specifications or verification guarantees.

AutoGen Multi-Agent Systems Enables complex agent interactions but lacks standardized protocol definitions and interoperability between different implementations.

Summary These approaches primarily focus on single-organization agent systems or simple client-server patterns, whereas BMPP addresses the more complex challenge of enabling verified interoperability between heterogeneous agent systems across organizational boundaries while maintaining the semantic richness necessary for effective LLM integration. **The focus of BSPL and BMPP is to provide an integration protocol for systems that may use all these different industry standards, it is definitely focused on systems interoperability so that engineers can use their system of choice while keeping the possibility of communicate with systems outside of their organisations safely, managing as much as possible all the well-known challenges [12] in current LLMs operability in sensitive and confidentiality-heavy industries.**

8 Conclusion and Future Work

BMPP, thanks to the work done on BSPL, successfully bridges the gap between formal protocol specification and natural language processing, enabling safe and performant integration of LLMs into distributed systems. The protocol maintains BSPL's formal verification guarantees while adding semantic richness that improves both machine and human interpretability.

Future work will focus on extending BMPP to support additional LLM architectures, optimizing performance for specific cloud platforms, and developing advanced verification techniques for natural language protocol specifications.

Acknowledgments

I thank the contributors to BSPL and MTP and the W3C Web Agents Working Group for their efforts that enabled this research.

References

[1] Singh, M. P. *Information-driven interaction-oriented programming: BSPL, a language and approach for choreographic development of distributed systems*. AAMAS 2011. Available at: https://www.cs.huji.ac.il/~jeff/aamas11/papers/A4_B57.pdf

[2] Vibelang Project. *Vibelang-rs*. Available at: https://github.com/Mec-iS/vibelang-rs

[3] Irugalbandara, C. *Meaning Typed Prompting: A Technique for Efficient, Reliable Structured Output Generation*. arXiv preprint arXiv:2410.18146, 2024. Available at: https://arxiv.org/abs/2410.18146

[4] *W3C Agents Features: Structured Generation Experiments*. Available at: https://github.com/Mec-iS/w3c-agents-features

[5] *BMPP Agents Rust Implementation*. Available at: https://github.com/Mec-iS/bmpp-agents-rs

[6] Yang, J., Jiang, D., He, L., Siu, S., Zhang, Y., Liao, D., Li, Z., Zeng, H., Jia, Y., Wang, H., Schneider, B., Ruan, C., Ma, W., Lyu, Z., Wang, Y., Lu, Y., Do, Q. D., Jiang, Z., Nie, P., Chen, W. *StructEval: Benchmarking LLMs' Capabilities to Generate Structural Outputs*. arXiv preprint arXiv:2505.20139, 2025. Available at: https://arxiv.org/abs/2505.20139

[7] Chopra, A. K. and Singh, M. P. *Correctness properties for multiagent systems*. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '08)*, pages 997–1004, 2008.

[8] Chopra, A. K. and Singh, M. P. *Producing compliant interactions: Conformance, coverage, and interoperability.* In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, pages 9–16, 2010.

[9] Chopra, A. K. and Singh, M. P. *Multiagent commitment alignment.* In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 937–944, 2009.

[10] *Features Engineering for LLMs contexts: W3C Web Agents Features Engineering Repository.* GitHub repository, 2025. Available at: https://github.com/Mec-iS/w3c-agents-features

[11] Dentella, V., Günther, F., Murphy, E., and Lackner, E. *Testing AI on language comprehension tasks reveals insensitivity to underlying meaning.* Scientific Reports, 14, 28083, 2024. DOI: https://doi.org/10.1038/s41598-024-79531-8 Available at: https://www.nature.com/articles/s41598-024-79531-8

[12] Shanmugarasa, Y., Pan, S., Ding, M., Zhao, D., and Rakotoarivelo, T. *Privacy Meets Explainability: Managing Confidential Data and Transparency Policies in LLM-Empowered Science.* arXiv preprint arXiv:2504.09961, 2025. Available at: https://arxiv.org/abs/2504.09961v1