Bridging Natural Language and Reservoir Simulation: A Configurable Architecture for Democratizing Subsurface Modeling

[Authors removed for blind review]
[Affiliation removed for blind review]

***Abstract*—Reservoir simulation remains one of the most technically demanding disciplines in the energy sector. Despite decades of software evolution, the fundamental barrier persists: translating engineering intent into executable simulation input requires specialized expertise that takes years to develop. This paper presents the architecture of the Oxy Reservoir Simulation Agent (ORSA), a system designed to bridge the gap between natural language interaction and domain-specific simulation syntax. We propose a multi-stage pipeline that decomposes translation into discrete, validated steps: speech recognition, intent classification, entity extraction, asset validation, syntax generation, and physics-based verification. The architecture addresses deployment heterogeneity through configuration-driven design, enabling air-gapped installations alongside cloud-native deployments. We present a domain mesh data architecture with purpose-built storage for different data types, and a hybrid synchronous-asynchronous communication pattern for long-running simulation workflows. This paper contributes reference architecture patterns applicable beyond reservoir simulation to any domain requiring natural language interfaces to complex technical systems.**

***Index Terms*—reservoir simulation, natural language processing, microservices architecture, domain-specific languages, configurable systems, event-driven architecture**

I. Introduction

Reservoir simulation represents a critical capability in hydrocarbon production optimization, enabling engineers to model fluid flow through porous media and predict production outcomes under various operational scenarios. The dominant simulation tools, including Schlumberger's ECLIPSE, require input files ('decks') written in complex domain-specific syntax with hundreds of interdependent keywords. Developing proficiency in these systems typically requires years of specialized training.

This expertise bottleneck creates organizational challenges. Simulation specialists become scarce resources, decisions requiring scenario analysis face delays, and institutional knowledge concentrates among a limited pool of practitioners. When experienced engineers retire or transition, organizations lose accumulated expertise that took decades to develop.

Previous attempts to address this challenge through graphical user interfaces have met limited success. The complexity of reservoir simulation resists reduction to form-based interaction; the combinatorial space of valid configurations exceeds what traditional GUI paradigms can meaningfully represent. Users inevitably encounter edge cases where the interface cannot express their intent, forcing fallback to manual deck editing.

Recent advances in natural language processing suggest an alternative approach. Rather than constraining users to predefined interface elements, systems can interpret natural language descriptions and generate appropriate technical syntax. This paper presents the Oxy Reservoir Simulation Agent (ORSA), a system architecture that translates conversational input directly into executable simulation configurations.

The technical challenges extend beyond language understanding. Global energy operations impose heterogeneous deployment requirements: some installations operate air-gapped with strict data sovereignty constraints, while others benefit from cloud-native infrastructure. The architecture must accommodate both extremes without fundamental redesign. Additionally, simulation runs range from seconds to hours, requiring careful separation of synchronous user interaction from asynchronous backend processing.

This paper makes the following contributions:

- A reference architecture for natural language interfaces to domain-specific technical systems
- Patterns for configuration-driven deployment across heterogeneous infrastructure
- Strategies for hybrid synchronous-asynchronous communication in long-running technical workflows
- A domain mesh approach to polyglot persistence in distributed systems

II. Related Work

A. Natural Language Interfaces for Technical Systems

Natural language interfaces to databases have received substantial attention since the 1970s [1]. Early systems like LUNAR [2] demonstrated feasibility for constrained domains. Modern approaches leverage transformer-based architectures for text-to-SQL translation [3], achieving competitive performance on benchmark datasets. However, reservoir simulation syntax presents distinct challenges: keyword interdependencies create constraints that SQL lacks, and physical consistency requirements demand validation beyond syntactic correctness.

B. Code Generation from Natural Language

Large language models have demonstrated remarkable capability in code generation tasks [4]. Systems like GitHub Copilot provide inline code completion, while research efforts explore more ambitious synthesis from high-level specifications [5]. Our work differs in targeting a constrained domain-specific language where outputs must satisfy physics-based consistency constraints beyond mere syntactic validity.

C. Microservices and Event-Driven Architecture

Microservices architecture has emerged as a dominant pattern for complex distributed systems [6]. Event-driven communication patterns address challenges of loose coupling and asynchronous processing [7]. Our architecture builds on these foundations while addressing domain-specific requirements for configurable deployment and long-running scientific computation.

III. System Architecture

The ORSA architecture comprises six primary layers, each addressing distinct concerns while maintaining loose coupling through well-defined interfaces. Fig. 1 illustrates the overall system structure.

A. Architectural Principles

Three principles guide architectural decisions throughout the system:

Configuration over convention: Every major component supports runtime configuration, enabling deployment-specific customization without code modification. Database backends, message brokers, model serving infrastructure, and authentication providers are all substitutable through configuration.

Domain-driven decomposition: Service boundaries align with domain concepts rather than technical layers. The simulation domain, training domain, and user management domain each own their data and business logic, communicating through events rather than shared databases.

Event-driven integration: Asynchronous messaging decouples services temporally. Long-running simulation jobs execute without blocking user interfaces, with progress updates delivered through event streams rather than polling.

B. User Interface Layer

The user interface layer supports multiple interaction modalities: voice input for hands-free operation in field environments, text chat for detailed technical discussions, web interfaces for visual feedback and result exploration, and REST APIs for programmatic integration. All modalities converge on a unified representation before entering the translation pipeline.

C. NLP Translation Layer

The translation layer converts natural language input to valid ECLIPSE deck syntax through a multi-stage pipeline (Fig. 2). Each stage performs a discrete transformation with explicit validation:

- **Speech Recognition:** Configurable ASR converts audio to text transcription with confidence scoring.
- **Intent Recognition:** Transformer-based classification identifies the user's operational goal.
- **Entity Extraction:** Named entity recognition identifies wells, dates, rates, and other domain objects.
- **Asset Validation:** Extracted entities are validated against the asset database to confirm existence and accessibility.
- **Syntax Generation:** A code generation model produces ECLIPSE keyword sequences from validated entities.
- **Deck Validation:** Physics-based validation ensures generated syntax respects reservoir engineering constraints.

Failed validation at any stage triggers rollback to the previous valid state. Low confidence scores prompt clarification requests rather than proceeding with uncertain interpretations.

IV. Data Architecture

The data layer implements a domain mesh pattern [8] with specialized storage for each bounded context (Fig. 3). This approach optimizes for access patterns while maintaining domain isolation.

A. Domain-Specific Storage

Five storage domains address distinct data characteristics:

Simulation Domain (Cassandra): Time-series simulation results and ECLIPSE deck files require high write throughput and horizontal scaling. Cassandra's columnar storage optimizes for append-heavy workloads with sequential read access patterns.

Training Domain (MongoDB): Machine learning datasets and model artifacts benefit from schema flexibility. Document storage accommodates evolving data structures without migration overhead.

User Domain (PostgreSQL): Authentication, authorization, and audit logging require ACID guarantees. Relational storage provides transactional integrity for security-critical operations.

Connectivity Domain (Neo4j): Well connectivity patterns and injection-production relationships form natural graph structures. Graph traversal queries efficiently answer questions about flow paths and interference patterns.

Cache Layer (Redis): Session state, frequently-accessed reference data, and computation results benefit from in-memory storage with sub-millisecond access latency.

B. Event-Driven Synchronization

Cross-domain data consistency is maintained through event sourcing rather than distributed transactions. Domain events published to a message broker (Kafka in high-throughput deployments, Redis Streams for lightweight installations) notify interested services of state changes. Each domain maintains eventual consistency with explicit conflict resolution policies.

V. Communication Architecture

The communication layer bridges synchronous frontend expectations with asynchronous backend processing (Fig. 4).

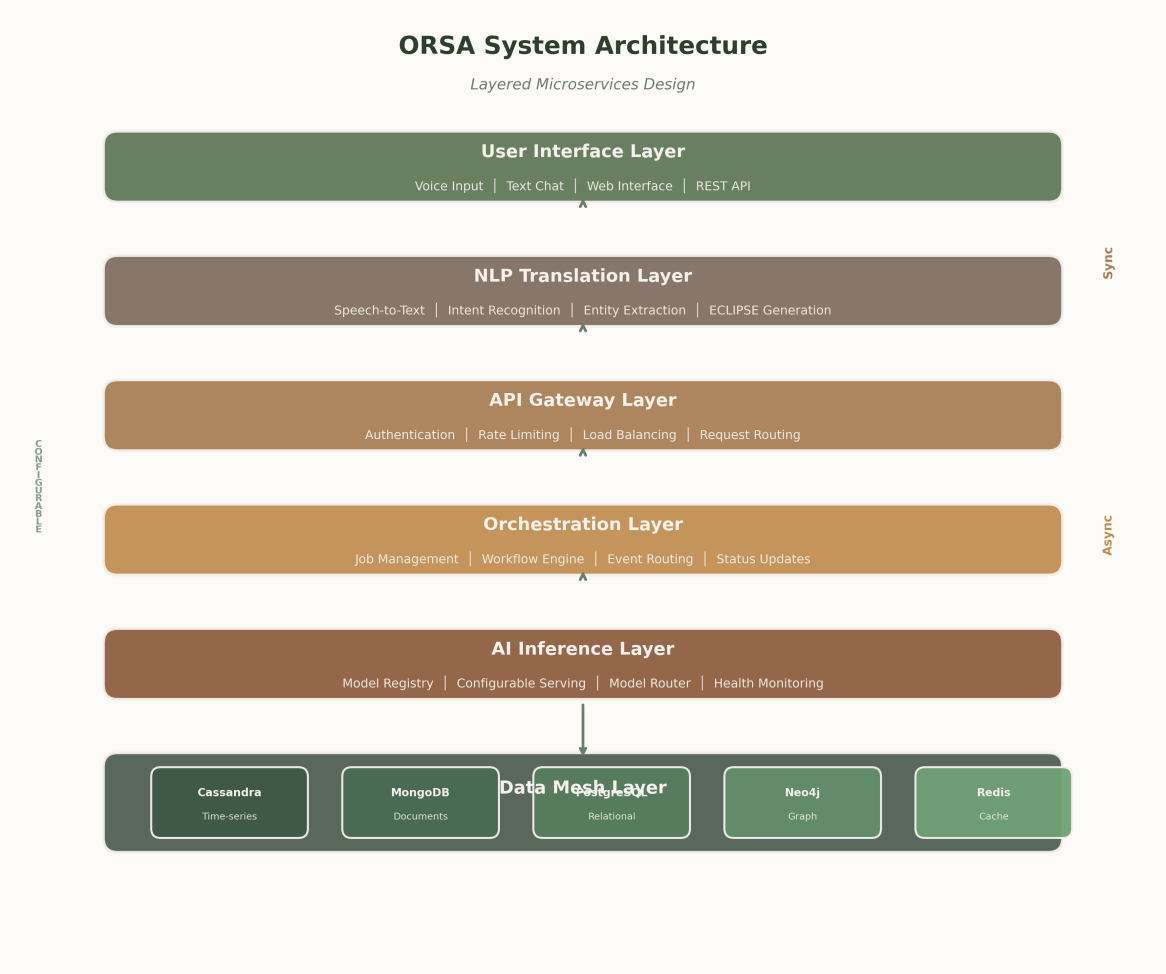

Fig. 1. ORSA System Architecture. The layered design separates concerns across User Interface, NLP Translation, API Gateway, Orchestration, AI Inference, and Data Mesh layers. Each layer is independently configurable for different deployment scenarios.

A. API Gateway

The API gateway provides unified access with authentication, rate limiting, and request routing. Synchronous endpoints handle quick operations (authentication, simple queries). Job submission endpoints return immediately with job identifiers, decoupling request acceptance from execution completion.

B. Orchestration Layer

A workflow orchestration engine manages job lifecycle: queuing, resource allocation, execution monitoring, and result delivery. WebSocket connections deliver real-time status updates to clients without polling. Failed jobs trigger configurable retry policies with exponential backoff.

C. Message Broker Configuration

The message broker abstraction supports multiple implementations: Apache Kafka for high-throughput production deployments, RabbitMQ for simpler installations, and Redis Streams for lightweight air-gapped environments. Selection occurs through configuration without code modification.

VI. AI Layer Design

A. Model Abstraction

The AI layer abstracts model access behind a unified interface, enabling runtime selection based on deployment constraints. Air-gapped installations run locally-hosted open-source models (CodeLlama, DeepSeek-Coder). Connected environments may leverage external APIs. Specialized deployments use custom fine-tuned models trained on proprietary simulation datasets.

B. Model Serving Infrastructure

Model serving infrastructure is similarly configurable: Ray Serve for high-performance multi-model deployments, Triton Inference Server for GPU-optimized production, or lightweight custom serving for resource-constrained environments. A model registry tracks versions, performance metrics, and health status, enabling automated failover when degradation is detected.

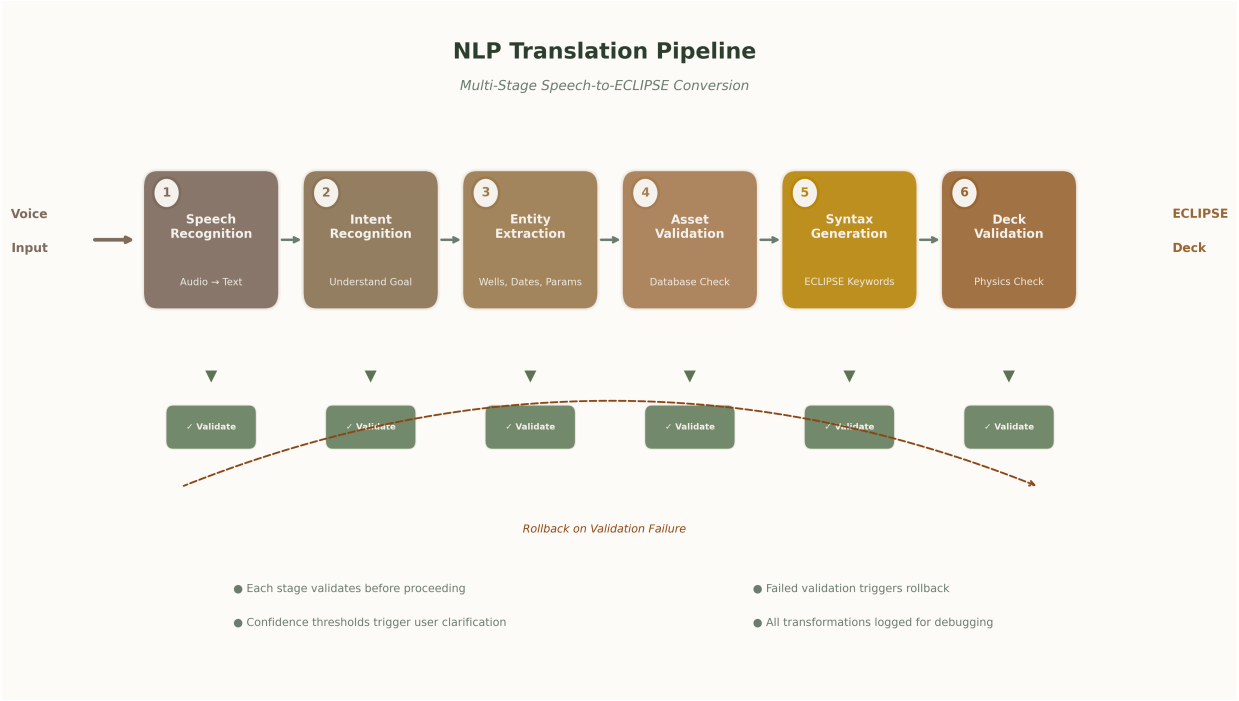

Fig. 2. NLP Translation Pipeline. The six-stage pipeline converts voice input to validated ECLIPSE deck syntax. Each stage includes validation checkpoints with rollback capability on failure.

C. Intelligent Routing

An AI-based router analyzes incoming requests and directs them to appropriate models based on complexity, required capabilities, and current system load. Simple queries route to lightweight models for fast response; complex physics validation routes to specialized models with domain expertise.

VII. Risk Assessment and Mitigation

Several architectural risks require explicit mitigation strategies:

NLP Translation Accuracy: Speech-to-ECLIPSE conversion must achieve high accuracy to be practically useful. The multi-stage pipeline with validation checkpoints provides graceful degradation—failed validation triggers clarification rather than incorrect output. Continuous model improvement from user feedback creates a virtuous cycle of increasing accuracy.

Model Synchronization: Multiple AI models must maintain consistent behavior across distributed deployments. A centralized model registry with version tracking, health monitoring, and automated failover addresses synchronization challenges.

Distributed Debugging: Asynchronous workflows across multiple services complicate debugging. Distributed tracing (Jaeger), centralized logging (ELK Stack), and comprehensive metrics (Prometheus/Grafana) provide observability across service boundaries.

Data Fragmentation: Polyglot persistence risks data inconsistency across domains. Event-driven synchronization with explicit data contracts and conflict resolution policies maintains eventual consistency without distributed transactions.

VIII. Deployment Considerations

A. Air-Gapped Environments

International operations frequently require air-gapped deployment with no external network connectivity. The architecture accommodates this through: locally-hosted models eliminating external API dependencies; self-contained infrastructure with all components deployable from local media; and offline update mechanisms using secure physical media transfer.

B. Cloud-Native Deployment

Cloud deployments leverage managed services for reduced operational burden: managed Kubernetes for container orchestration, managed databases for persistence, and auto-scaling for elastic capacity. The same application code runs in both environments; only infrastructure configuration differs.

C. Security Architecture

Security requirements span authentication (pluggable identity providers), authorization (role-based access control), encryption (TLS in transit, encryption at rest), and audit logging (immutable audit trails for compliance). The architecture implements defense in depth with security controls at each layer.

IX. Conclusions

This paper presented the architecture of ORSA, a system that bridges natural language interaction and reservoir simulation through configurable, event-driven microservices. The

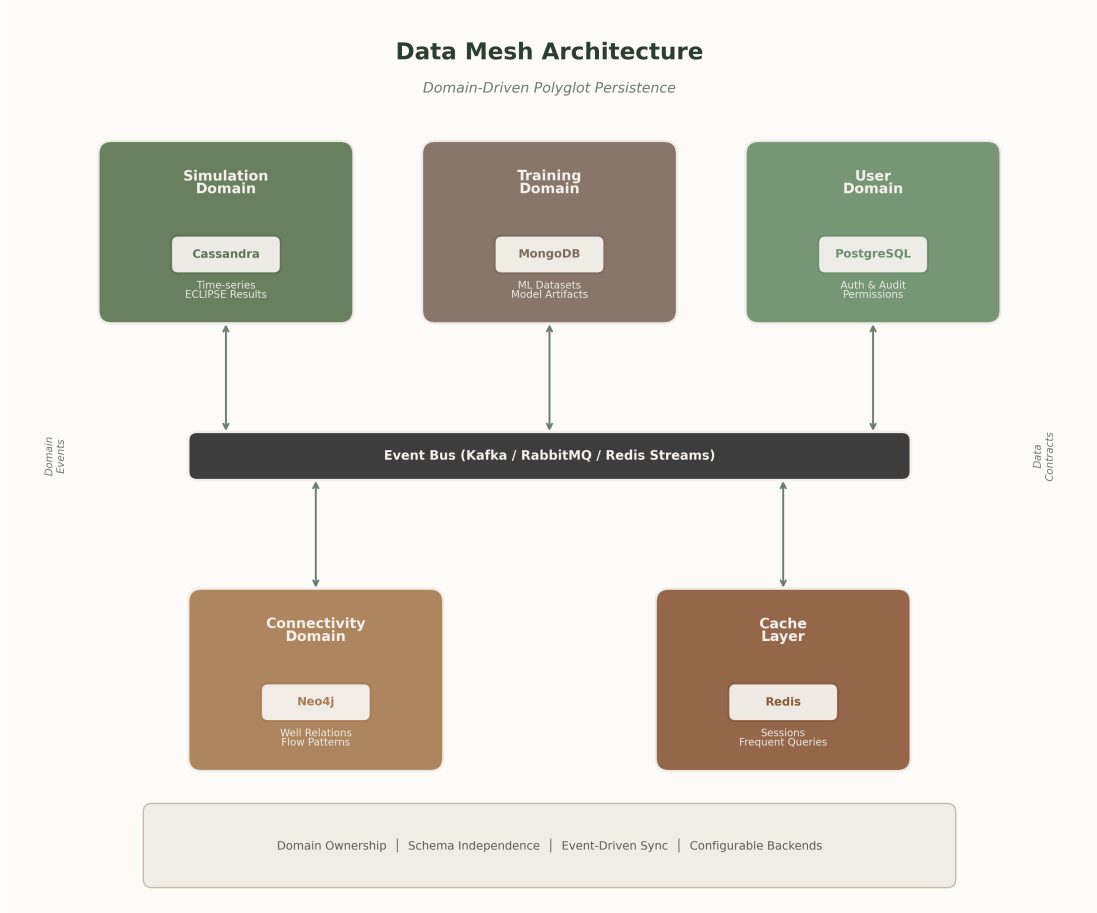

Fig. 3. Data Mesh Architecture. Five specialized storage domains communicate through a central event bus, maintaining eventual consistency through event-driven synchronization.

multi-stage NLP pipeline decomposes the translation challenge into validated steps, enabling graceful degradation when confidence is insufficient. The domain mesh data architecture optimizes storage for different data types while maintaining eventual consistency through event sourcing.

The configuration-driven approach addresses deployment heterogeneity inherent in global energy operations, supporting both air-gapped installations and cloud-native deployments from a common codebase. This flexibility positions the architecture for long-term evolution as AI capabilities advance—new models can be integrated through configuration rather than architectural revision.

Beyond reservoir simulation, the architectural patterns presented here apply to any domain requiring natural language interfaces to complex technical systems. The combination of multi-stage translation pipelines, configurable infrastructure, and event-driven communication provides a template for democratizing access to specialized technical capabilities.

Future work includes quantitative evaluation of translation accuracy across diverse query types, user studies comparing natural language interaction to traditional interfaces, and extension to additional simulation platforms beyond ECLIPSE.

References

[1] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases – an introduction," *Natural Language Engineering*, vol. 1, no. 1, pp. 29–81, 1995.

[2] W. A. Woods, "Progress in natural language understanding: an application to lunar geology," in *Proc. AFIPS National Computer Conference*, 1973, pp. 441–450.

[3] T. Yu *et al.*, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," in *Proc. EMNLP*, 2018, pp. 3911–3921.

[4] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[5] Y. Li *et al.*, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[6] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly Media, 2021.

[7] M. Fowler, "Event-driven architecture," martinfowler.com, 2017. [Online]. Available: https://martinfowler.com/articles/201701-event-driven.html

[8] Z. Dehghani, "Data mesh principles and logical architecture," martinfowler.com, 2020. [Online]. Available: https://martinfowler.com/articles/data-mesh-principles.html

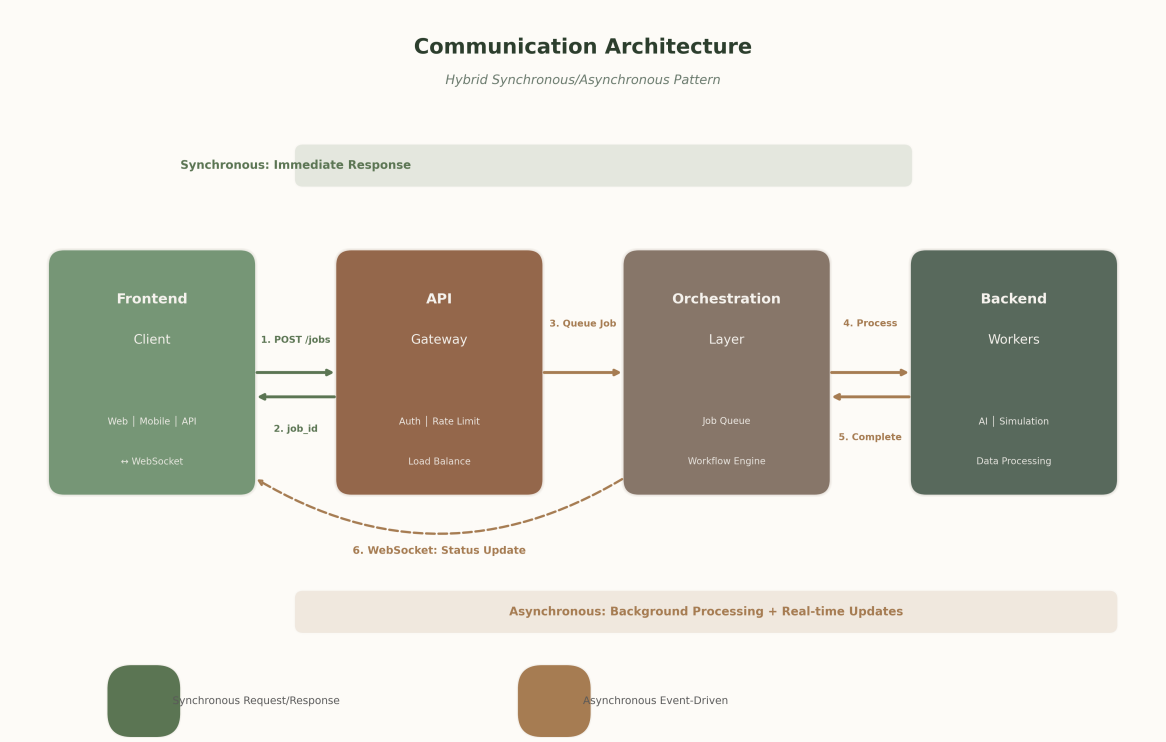

Fig. 4. Communication Architecture. The hybrid synchronous-asynchronous pattern provides immediate response to users while processing long-running jobs in the background with real-time status updates via WebSocket.