Project JANUS

Neuromorphic Trading Intelligence

Complete Documentation Suite

A Brain-Inspired Architecture for Autonomous Financial Systems

Documentation Contents

1. **Main Architecture** — System design, philosophical foundation, and integration strategy
2. **Forward Service** — Real-time decision-making, pattern recognition, and execution
3. **Backward Service** — Memory consolidation, schema formation, and learning
4. **Neuromorphic Architecture** — Brain-region mapping and component design
5. **Rust Implementation** — Production-ready system with deployment guide

Author

Jordan Smith

Date

December 26, 2025

“The god of beginnings and transitions, looking simultaneously to the future and the past.”

How to Use This Document

This unified document combines all five technical specifications of Project JANUS into a single, comprehensive reference. Each part maintains its original formatting and can be referenced independently.

Reading Paths

For Quantitative Researchers

1. Start with **Part 1: Main Architecture** for system overview
2. Read **Part 2: Forward Service** for trading algorithms
3. Review **Part 3: Backward Service** for learning mechanisms

For ML Engineers

1. Begin with **Part 5: Rust Implementation** for tech stack
2. Study **Part 2: Forward Service** for model architecture
3. Explore **Part 3: Backward Service** for training pipeline

For System Architects

1. Review **Part 1: Main Architecture** for system design
2. Read **Part 4: Neuromorphic Architecture** for component mapping
3. Study **Part 5: Rust Implementation** for deployment strategy

For Neuroscience Enthusiasts

1. Start with **Part 4: Neuromorphic Architecture** for brain mapping
2. Read **Part 1: Main Architecture** for philosophical motivation
3. Explore **Parts 2 & 3** for biological implementation details

Note on Technical Depth

This documentation is highly technical and assumes familiarity with:

- Machine learning (neural networks, reinforcement learning)
- Neuroscience concepts (hippocampus, basal ganglia, sharp-wave ripples)
- Quantitative finance (market microstructure, optimal execution)
- Systems programming (Rust, async architectures, distributed systems)

Document Information

- **Total Pages:** Approximately 200+ pages
- **Combined Size:** All five technical specifications in one file
- **Format:** Each part preserves its original table of contents and section numbering
- **Repository:** `https://github.com/nuniesmith/technical_papers`

Project JANUS

Neuromorphic Trading Intelligence

A Brain-Inspired Architecture for Autonomous Financial Systems

The Two-Faced Architecture

Forward (Janus Bifrons): The Conscious Trader
Visual perception, logical reasoning, and real-time execution

Backward (Janus Consivius): The Sleeping Mind
Memory consolidation, schema formation, and learning

Classification: Main Architecture Document

Version: 2.0

Author: Jordan Smith
github.com/nuniesmith

Date: December 26, 2025

"The god of beginnings and transitions, looking simultaneously to the future and the past."

Abstract

Financial markets have evolved into complex adaptive systems operating at timescales far beyond human perception. Modern high-frequency trading requires decision-making in microseconds, processing millions of data points across multiple modalities—price movements, order flow toxicity, news sentiment, and macroeconomic signals—while adhering to strict regulatory and risk management constraints. The challenge is not merely computational speed, but the integration of *perception*, *reasoning*, and *memory* into a unified autonomous system.

This document presents **Project JANUS**, a neuromorphic trading intelligence system inspired by the bifurcated nature of its namesake—the Roman god who simultaneously looks forward and backward. JANUS represents a paradigm shift from monolithic deep learning models to a brain-inspired architecture that mirrors the functional specialization and information flow patterns observed in biological neural systems.

The architecture is fundamentally dual:

- **JANUS Forward (Janus Bifrons)**: The "wake state" trading engine that operates in real-time market conditions. It implements visual pattern recognition through Gramian Angular Fields (GAF) and Video Vision Transformers (ViViT), symbolic reasoning through Logic Tensor Networks (LTN), multimodal fusion via gated cross-attention, and neuromorphic decision-making inspired by basal ganglia dual pathways.
- **JANUS Backward (Janus Consivius)**: The "sleep state" memory consolidation system that processes trading experiences during off-market hours. It implements a three-timescale memory hierarchy (hippocampal episodic buffer, sharp-wave ripple replay, and neocortical schema formation), UMAP-based cognitive visualization, and integration with vector databases for long-term knowledge storage.

By separating the hot-path real-time inference (Forward) from the cold-path batch learning (Backward), JANUS achieves both microsecond-latency execution and deep, contemplative learning—mirroring the wake-sleep cycle of biological brains. The system is implemented in Rust for performance-critical components and Python for training pipelines, with explicit neuromorphic mapping to brain regions: visual cortex, prefrontal cortex, hippocampus, basal ganglia, thalamus, amygdala, hypothalamus, cerebellum, and integration centers.

This main document provides the conceptual framework, architectural philosophy, and system integration overview. Detailed technical specifications are provided in companion documents:

- `janus_forward.tex` — Forward service algorithms and implementation

- `janus_backward.tex` — Backward service memory architecture
- `janus_neuromorphic_architecture.tex` — Brain region mapping and neuromorphic design
- `janus_rust_implementation.tex` — Rust implementation strategy and deployment

Key Contributions:

1. A neuromorphic architecture that maps trading functions to specialized brain regions
2. Differentiable Gramian Angular Fields (DiffGAF) for learnable time series imaging
3. Logic Tensor Networks for hard constraint enforcement in financial decision-making
4. Sharp-wave ripple simulation for experience replay with 10-20$\times$ time compression
5. Dual-service architecture separating real-time inference from batch consolidation
6. Rust-first implementation for safety-critical financial systems

Contents

1 Introduction: The Crisis of Complexity

1.1 The Evolution of Quantitative Trading

The history of quantitative trading can be characterized by escalating complexity and decreasing human interpretability:

- **Quant 1.0 (1970s-1990s):** Rule-based expert systems and technical indicators (moving averages, RSI, Bollinger Bands). Human-interpretable but brittle and unable to capture complex non-linear dynamics.
- **Quant 2.0 (1990s-2010s):** Statistical arbitrage and factor models (ARIMA, GARCH, Fama-French factors). Economically grounded but limited by linear assumptions and Gaussian priors.
- **Quant 3.0 (2010s-2020s):** Deep learning and end-to-end optimization (LSTMs, Transformers, Deep RL). Powerful pattern recognition but opaque reasoning, regulatory risks, and catastrophic failures during regime shifts.
- **Quant 4.0 (2020s-present):** Neuro-symbolic systems combining neural perception with symbolic reasoning. The integration of vision-based pattern recognition, logical constraint satisfaction, and hierarchical decision-making.

Project JANUS represents a realization of the Quant 4.0 vision, moving beyond pure deep learning toward hybrid systems that combine the intuitive power of neural networks with the logical rigor of symbolic AI and the biological plausibility of neuromorphic architectures.

1.2 The Black Box Crisis

Modern deep reinforcement learning agents can achieve superhuman performance on complex tasks, but this performance comes at a cost: *opacity*. A DRL agent trained solely to maximize Sharpe ratio may discover strategies that:

- Exploit simulator artifacts that don't exist in live markets (reality gap)
- Violate regulatory constraints (wash sales, market manipulation, front-running)
- Fail catastrophically during regime shifts unseen in training data
- Exhibit emergent behaviors that are impossible to audit or explain

In traditional software engineering, this would be unacceptable. Financial systems require:

1. **Explainability:** Every trading decision must be auditable and defensible
2. **Safety:** Hard constraints (risk limits, regulatory rules) must be enforced
3. **Robustness:** Systems must degrade gracefully under extreme market conditions
4. **Adaptability:** Systems must learn from experience without catastrophic forgetting

The challenge is to build systems that achieve these properties while maintaining the pattern recognition power of modern deep learning.

1.3 The Neuromorphic Solution

Biological brains solve many of the same challenges faced by autonomous trading systems:

- **Real-time processing:** Sensory-motor loops operate in milliseconds
- **Multi-timescale learning:** Immediate reactions (reflexes), medium-term adaptation (skill learning), long-term consolidation (memory)
- **Constraint satisfaction:** Motor commands must respect physical constraints (joint limits, balance)
- **Homeostasis:** Internal states (hunger, stress) must be regulated
- **Memory consolidation:** Experiences are replayed and abstracted during sleep

By mapping trading functions to brain regions with analogous roles, JANUS creates a system that is both neuroscientifically inspired and functionally specialized:

This mapping is not merely metaphorical—it guides the architectural design, data flow, and optimization strategies throughout the system.

Brain Region	Neuroscience Role	Trading Role
Visual Cortex	Pattern recognition in images	GAF/ViViT market pattern detection
Prefrontal Cortex	Logic, planning, rule adherence	LTN constraint enforcement
Hippocampus	Episodic memory, replay	Experience buffer, SWR simulation
Neocortex	Long-term schemas	Consolidated strategies, vector DB
Basal Ganglia	Action selection, Go/No-Go	Dual-pathway decision engine
Cerebellum	Motor control, prediction	Order execution, market impact model
Thalamus	Attention, multimodal fusion	Gated cross-attention fusion
Amygdala	Fear, threat detection	Circuit breakers, risk alerts
Hypothalamus	Homeostasis, drive states	Risk appetite, position sizing

Table 1: Neuromorphic Mapping: Brain Regions to Trading Functions

2 Architectural Philosophy: The Two Faces of JANUS

2.1 Why Dual Architecture?

The central insight of Project JANUS is that *real-time decision-making* and *reflective learning* are fundamentally different computational regimes that should be decoupled:

- **Forward (Wake State):** Operates during market hours under extreme latency constraints (microseconds to milliseconds). Must process streaming data, fuse multimodal inputs, evaluate logical constraints, and execute trades—all while maintaining deterministic worst-case performance.
- **Backward (Sleep State):** Operates during off-market hours without latency constraints. Can perform computationally expensive operations: prioritized experience replay, UMAP dimensionality reduction, schema clustering, vector database consolidation, and model retraining.

This separation mirrors the biological wake-sleep cycle, where:

- During wakefulness, the brain prioritizes fast sensory-motor integration
- During sleep, the brain replays experiences (sharp-wave ripples in hippocampus), consolidates memories to neocortex, and prunes synapses

2.2 Janus Bifrons: The Forward Face

Janus Bifrons ("two-faced") represents the conscious, awake trader. The Forward service implements:

1. **Visual Pattern Recognition:**
    - Differentiable Gramian Angular Fields (DiffGAF) transform 1D time series into 2D images with learnable normalization
    - GAF Video sequences capture spatiotemporal market evolution
    - Video Vision Transformer (ViViT) processes multi-frame sequences with factorized spatial-temporal attention
    - Limit Order Book (LOB) heatmap fusion for microstructure awareness
2. **Symbolic Reasoning:**
    - Logic Tensor Networks (LTN) embed regulatory and risk constraints as differentiable logical predicates
    - Lukasiewicz T-Norm operations (AND, OR, NOT, IMPLIES, FORALL, EXISTS) enable gradient-based satisfaction
    - Knowledge base includes wash sale rules, Almgren-Chriss risk constraints, and VPIN toxicity thresholds
3. **Multimodal Fusion:**
    - Gated Cross-Attention fuses visual embeddings (ViViT), time series forecasts (Chronos-Bolt), and text embeddings (FinBERT)
    - Learnable gating weights determine modality importance dynamically
4. **Neuromorphic Decision Engine:**
    - Basal ganglia-inspired dual pathways: Direct (Go) and Indirect (No-Go)
    - Cerebellar forward model predicts market impact and execution cost
    - Action is authorized only when Go > No-Go and LTN constraints are satisfied

2.3 Janus Consivius: The Backward Face

Janus Consivius (“sower”, “planter”) represents the reflective, consolidating mind. The Backward service implements:

1. **Three-Timescale Memory Hierarchy:**
    - *Short-term (Hippocampus):* Episodic buffer stores raw experiences with pattern separation and sparse encoding
    - *Medium-term (SWR):* Sharp-wave ripple simulation replays high-priority experiences with 10-20$\times$ time compression

- *Long-term (Neocortex):* Schema formation via clustering and consolidation to vector database (Qdrant)

2. **Prioritized Replay:**
   - Experiences prioritized by TD-error, LTN violation severity, and recency
   - SumTree data structure for $O(\log N)$ sampling
   - Importance sampling correction to prevent bias
3. **Cognitive Visualization:**
   - AlignedUMAP projects experiences across training epochs to detect schema formation
   - Parametric UMAP enables real-time anomaly detection and regime shift visualization
4. **Vector Database Integration:**
   - Qdrant stores consolidated schemas as high-dimensional embeddings
   - Similarity search retrieves relevant past experiences for few-shot adaptation
   - Periodic pruning removes outdated or redundant schemas

2.4 Information Flow Between Services

The Forward and Backward services communicate asynchronously:

1. **During Market Hours (Forward Active):**
   - Forward service processes live market data and executes trades
   - Experiences (state, action, reward, next state, LTN violations) are written to shared episodic buffer
   - No blocking—buffer writes are lock-free and wait-free
2. **During Off-Hours (Backward Active):**
   - Backward service consumes episodic buffer
   - Prioritized replay generates training batches
   - SWR simulation compresses experiences
   - Schemas are consolidated to Qdrant
   - Updated model weights are exported (ONNX) and versioned
3. **Model Deployment:**

- Forward service loads new ONNX models during market close
- Shadow deployment allows parallel evaluation before promotion
- Graceful fallback to previous model if validation fails

3 Core Components: Hybrid Intelligence

3.1 Vision: Seeing the Market's Geometry

3.1.1 Why Visual Encoding?

Financial time series are traditionally represented as 1D sequences of scalars. This representation is efficient for storage but discards the rich topological structure of market dynamics. By transforming time series into images, we unlock the architectural power of Computer Vision:

- **Convolutional layers** detect hierarchical spatial patterns (edges, textures, shapes)
- **Translation invariance** recognizes patterns regardless of position
- **Attention mechanisms** focus on salient regions

In the context of transformed time series, these visual patterns correspond to:

- *Edges* $\rightarrow$ Regime transitions (trend reversals, volatility shifts)
- *Textures* $\rightarrow$ Microstructure patterns (volatility clustering, mean reversion)
- *Shapes* $\rightarrow$ Macrostructure formations (head-and-shoulders, flags, triangles)

3.1.2 Differentiable Gramian Angular Fields (DiffGAF)

JANUS introduces **DiffGAF**, a learnable variant of Gramian Angular Fields that enables end-to-end gradient flow from the visual classifier back through the imaging transformation.

Given a time series $X = \{x_1, x_2, \ldots, x_n\}$:

Step 1: Learnable Normalization

$$\tilde{x}_i = \tanh\left(\frac{x_i - \min(X)}{\max(X) - \min(X)} \cdot \alpha + \beta\right) \tag{1}$$

where α, β are learnable parameters optimized during training.

Step 2: Polar Encoding

$$\begin{cases} \phi_i = \arccos(\tilde{x}_i), & -1 \leq \tilde{x}_i \leq 1 \\ r_i = \frac{t_i}{N}, & t_i \in \mathbb{N} \end{cases} \tag{2}$$

Step 3: Gramian Field Generation

Gramian Angular Summation Field (GASF):

$$\text{GASF}_{i,j} = \cos(\phi_i + \phi_j) \tag{3}$$

Gramian Angular Difference Field (GADF):

$$\text{GADF}_{i,j} = \sin(\phi_i - \phi_j) \tag{4}$$

The resulting $n \times n$ image preserves temporal correlation structure while enabling spatial convolution.

3.1.3 GAF Video and ViViT

Static images capture instantaneous market state. To capture *evolution*, JANUS generates GAF video sequences using sliding windows:

$$\mathcal{V} = \{GAF(X_{t:t+w}), GAF(X_{t+s:t+w+s}), \ldots\} \tag{5}$$

where w is window size and s is stride. The resulting 3D tensor $\mathcal{V} \in \mathbb{R}^{F \times H \times W}$ is processed by a Video Vision Transformer (ViViT) with:

- **Spatial attention** within each frame (volatility clusters, trend structures)
- **Temporal attention** across frames (regime transitions, momentum shifts)

3.2 Logic: Enforcing the Rules of the Game

3.2.1 The Necessity of Symbolic Constraints

Deep learning excels at pattern recognition but lacks logical precision. In finance, "close enough" is unacceptable:

- A wash sale (selling at a loss and repurchasing within 30 days) triggers tax penalties
- Exceeding risk limits violates Almgren-Chriss constraints and regulatory mandates
- Order flow toxicity (high VPIN) indicates informed trading and adverse selection risk

These are not soft preferences—they are *hard constraints* that must be satisfied with mathematical certainty.

3.2.2 Logic Tensor Networks (LTN)

LTN embeds First-Order Logic (FOL) into neural networks by grounding logical symbols in continuous tensor space:

- **Constants** $\rightarrow$ Tensor embeddings
- **Predicates** $\rightarrow$ Neural networks outputting $[0, 1]$ truth values
- **Logical connectives** $\rightarrow$ Fuzzy logic t-norms

JANUS uses Lukasiewicz t-norms for improved gradient flow:

$$\text{AND}(p, q) = \max(0, p + q - 1) \tag{6}$$
$$\text{OR}(p, q) = \min(1, p + q) \tag{7}$$
$$\text{NOT}(p) = 1 - p \tag{8}$$
$$\text{IMPLIES}(p, q) = \min(1, 1 - p + q) \tag{9}$$

3.2.3 Knowledge Base Examples

Wash Sale Constraint:

$$\forall t : \text{Sold}(t) \wedge \text{Loss}(t) \implies \neg\text{Buy}(t, t + 30) \tag{10}$$

"If a position was sold at a loss, do not repurchase within 30 days."
Almgren-Chriss Risk Constraint:

$$\forall \tau : \text{Variance}(\tau) \leq \lambda \cdot \text{MarketImpact}(\tau) \tag{11}$$

"Execution variance must not exceed risk tolerance."
VPIN Toxicity Constraint:

$$\text{VPIN}(t) > \theta \implies \text{Widen}(\text{spread}) \vee \text{Halt}(\text{trading}) \tag{12}$$

"If order flow toxicity exceeds threshold, widen spreads or halt."

3.3 Fusion: Integrating Multiple Realities

3.3.1 The Multimodal Challenge

Markets are inherently multimodal. A complete understanding requires:

- **Visual patterns** (GAF video, LOB heatmaps)

- **Time series forecasts** (Chronos-Bolt probabilistic predictions)
- **Text semantics** (news, earnings calls, social media)
- **Logical constraints** (LTN predicate evaluations)

The challenge is not merely concatenating these modalities but learning their relative importance dynamically.

3.3.2 Gated Cross-Attention (GCA)

JANUS uses GCA to fuse modalities with learnable gating:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{13}$$

$$g = \sigma(W_g \cdot [x_{\text{visual}}; x_{\text{time}}; x_{\text{text}}] + b_g) \tag{14}$$

$$x_{\text{fused}} = g \odot \text{Attention}(x_{\text{visual}}, x_{\text{time}}, x_{\text{time}}) \tag{15}$$

The gating weight $g \in [0, 1]$ determines how much to trust each modality. During high volatility, visual patterns may dominate; during earnings season, text embeddings may be prioritized.

3.4 Decision: The Neuromorphic Motor System

3.4.1 Basal Ganglia Dual Pathways

The basal ganglia implements action selection via competing pathways:

- **Direct Pathway (Go):** Facilitates action execution
- **Indirect Pathway (No-Go):** Inhibits action execution

JANUS mirrors this with dual neural networks:

$$\text{Go}(s) = f_{\text{direct}}(s; \theta_{\text{Go}}) \tag{16}$$

$$\text{NoGo}(s) = f_{\text{indirect}}(s; \theta_{\text{NoGo}}) \tag{17}$$

$$a = \begin{cases} a_{\text{proposed}} & \text{if Go}(s) > \text{NoGo}(s) \wedge \text{LTN}(s, a) > \tau \\ \text{HOLD} & \text{otherwise} \end{cases} \tag{18}$$

This architecture naturally implements risk-averse behavior: actions are blocked unless both pathways agree AND logical constraints are satisfied.

3.4.2 Cerebellar Forward Model

The cerebellum predicts sensory consequences of motor commands. JANUS implements a market impact predictor:

$$\hat{p}_{\text{fill}} = f_{\text{cerebellum}}(a, s_{\text{LOB}}) \tag{19}$$

This prediction enables:

- **Trajectory adjustment:** Modify order size/timing to minimize slippage
- **Error correction:** Compare predicted vs. actual fill price and update model

4 Memory and Learning: The Sleeping Mind

4.1 The Three-Timescale Hierarchy

Biological memory systems operate at multiple timescales:

- **Hippocampus:** Rapid encoding of episodic details (seconds to hours)
- **Sharp-Wave Ripples:** Replay and consolidation during rest (minutes to hours)
- **Neocortex:** Long-term schemas and abstract knowledge (days to years)

JANUS replicates this hierarchy:

4.1.1 Short-Term: Hippocampal Episodic Buffer

Raw experiences are stored with minimal processing:

$$e_t = (s_t, a_t, r_t, s_{t+1}, \text{done}_t, \text{LTN}_t, \text{meta}_t) \tag{20}$$

where LTN_t records constraint violations and meta_t includes timestamps, market regime labels, etc.

Pattern separation ensures diverse storage:

$$\text{similarity}(e_i, e_j) < \tau_{\text{sep}} \implies \text{store both} \tag{21}$$

4.1.2 Medium-Term: Sharp-Wave Ripple (SWR) Simulation

During sleep, the hippocampus replays experiences at 10-20$\times$ real-time speed. JANUS simulates this with:

1. **Prioritized sampling:** Select high TD-error, high LTN-violation experiences
2. **Time compression:** Process entire trading day in 10-30 minutes
3. **Batch generation:** Create training batches for model updates

Priority weighting:

$$p_i = (|\delta_i| + \epsilon)^\alpha + \beta \cdot \text{LTN\_violation}_i + \gamma \cdot \text{recency}_i \tag{22}$$

4.1.3 Long-Term: Neocortical Schemas

Repeated experiences are abstracted into schemas—general patterns that transcend specific episodes:

$$\theta_{\text{schema}} \leftarrow \theta_{\text{schema}} + \eta \cdot \text{recall\_gate} \cdot \nabla_\theta \mathcal{L} \tag{23}$$

Recall gating prevents interference:

$$\text{recall\_gate} = \mathbb{1}[\text{cosine\_similarity}(e_{\text{current}}, \text{schema}) > \tau_{\text{recall}}] \tag{24}$$

Only experiences similar to existing schemas contribute to consolidation, reducing catastrophic forgetting.

4.2 Cognitive Visualization: UMAP

4.2.1 AlignedUMAP for Schema Detection

During training, experiences are projected to 2D space across epochs:

$$\mathcal{L}_{\text{UMAP}} = \sum_{i,j} w_{ij} \log\left(\frac{1}{1 + \|y_i - y_j\|^2}\right) + (1 - w_{ij}) \log\left(1 - \frac{1}{1 + \|y_i - y_j\|^2}\right) \tag{25}$$

Cluster formation in UMAP space indicates emerging schemas. Cluster stability across epochs validates consolidation.

4.2.2 Parametric UMAP for Anomaly Detection

A neural network learns the UMAP projection:

$$y = f_{\text{UMAP}}(x; \theta) \tag{26}$$

At inference time, new experiences are projected in real-time. Outliers indicate:

- Regime shifts (market structure change)
- Novel strategies (unexplored state space)
- Data quality issues (sensor failures)

5 Implementation Strategy: Rust-First Architecture

5.1 Why Rust?

Financial systems demand:

- **Performance:** Microsecond latencies, zero-copy operations
- **Safety:** No null pointers, no data races, no undefined behavior
- **Reliability:** Predictable performance, deterministic memory usage
- **Auditability:** Strong type system, explicit error handling

Rust provides all of these without garbage collection pauses or runtime overhead.

5.2 Service Architecture

JANUS is implemented as two Rust services with a Python training gateway:

1. **Forward Service (Rust):**
   - Async runtime (Tokio) for concurrent market data streams
   - ONNX Runtime for model inference (ViViT, LTN predicates)
   - Lock-free queues for experience buffer writes
   - gRPC API for order submission
2. **Backward Service (Rust):**
   - Rayon for parallel batch processing
   - SumTree (custom implementation) for prioritized replay
   - Qdrant client for vector database operations
   - UMAP projection (via ONNX or direct Rust port)
3. **Training Gateway (Python):**
   - PyTorch for model training
   - FastAPI for REST endpoints
   - Celery for async batch jobs (SWR simulation, model export)
   - Model export to ONNX for Rust consumption

5.3 ML Framework Migration Path

1. **Phase 1 (Months 1-3):** Hybrid PyTorch + ONNX
   - Train in PyTorch, export to ONNX, infer in Rust
   - Establish benchmarks and validation pipelines
2. **Phase 2 (Months 4-6):** Rust-native inference
   - Port inference to tch-rs (libtorch) or Candle
   - Eliminate Python dependency for deployment
3. **Phase 3 (Months 7-12):** Full Rust ML stack
   - Port training to Burn or Candle
   - Single-language codebase for maximum auditability

5.4 Deployment Architecture

Development:

- Docker Compose for local multi-service orchestration
- Shared volumes for model artifacts and experience buffers
- PostgreSQL for metadata, Qdrant for vectors, Redis for pub/sub

Production:

- Kubernetes for orchestration and auto-scaling
- StatefulSets for Qdrant and PostgreSQL
- Helm charts for versioned deployments
- Prometheus + Grafana for monitoring
- Shadow deployment for A/B testing

6 Safety and Compliance: The Glass Box

6.1 Architectural Invariants

JANUS enforces architectural invariants through Rust's type system:

1. **No Panics in Hot Path:**
   - All errors are typed (using `thiserror`)
   - `unwrap()` and `expect()` forbidden in production code
   - Fallback strategies for all error conditions
2. **LTN Constraints Always Evaluated:**
   - Actions cannot be executed without LTN evaluation
   - Constraint violations logged and traced
   - Circuit breakers halt trading on repeated violations
3. **Memory Bounds Enforced:**
   - Episodic buffer has maximum size with FIFO eviction
   - No unbounded allocations in hot path
   - Pre-allocated buffers for latency-critical operations
4. **Temporal Guarantees:**
   - SWR compression factor $\in [10, 20]$ verified at runtime
   - Replay cannot exceed configured wall-clock time
   - Timeout guards on all external API calls

6.2 Explainability and Auditability

Every trading decision generates an audit trail:

1. **Visual Attention:** Grad-CAM heatmaps show which GAF frames influenced the decision
2. **Logical Trace:** LTN evaluations are logged with predicate truth values
3. **Modality Weights:** GCA gating values indicate which inputs were trusted
4. **Pathway Activation:** Go/No-Go scores explain action selection
5. **Forward Model Error:** Predicted vs. actual slippage quantifies model confidence

This trace is stored in structured logs (JSON) and indexed for regulatory queries.

6.3 Circuit Breakers and Kill Switches

The **Amygdala** subsystem implements safety-critical overrides:

1. **Volatility Spike:** Halt trading if realized volatility $> 3\times$ historical
2. **Drawdown Limit:** Halt if portfolio drawdown $> 10\%$ in single session
3. **LTN Violation Rate:** Halt if $> 5\%$ of actions violate constraints
4. **Execution Anomaly:** Halt if average slippage $> 2\times$ predicted
5. **Manual Override:** Human operators can trigger emergency halt via API

All circuit breakers are *fail-safe*: they default to halting trading on error or uncertainty.

7 Validation and Testing: Proving Robustness

7.1 Simulation and Backtesting

JANUS is validated through progressive realism:

1. **Unit Tests:** Each component tested in isolation
   - GAF transformation invertibility
   - LTN predicate gradient flow
   - SumTree priority sampling correctness
2. **Synthetic Markets:** Controlled environments with known properties
   - Mean-reverting Ornstein-Uhlenbeck process
   - Momentum-driven geometric Brownian motion
   - Regime-switching models with abrupt transitions
3. **Historical Replay:** Real market data with simulated execution
   - Out-of-sample testing on unseen time periods
   - Walk-forward optimization to prevent overfitting
   - Comparison with baseline strategies (buy-and-hold, momentum, mean-reversion)
4. **Black Swan Stress Tests:** Extreme scenarios
   - Flash crash (May 6, 2010)
   - COVID crash (March 2020)
   - GameStop short squeeze (January 2021)
5. **Paper Trading:** Live market data with simulated execution
   - Reality gap monitoring (predicted vs. actual slippage)
   - Latency profiling under production load
6. **Shadow Deployment:** Parallel execution with production system
   - JANUS generates signals but does not execute
   - Performance compared with existing live system
   - Gradual rollout (1% $\rightarrow$ 10% $\rightarrow$ 50% $\rightarrow$ 100%)

7.2 Comparative Benchmarks

JANUS is compared against:

- **Baseline:** Buy-and-hold, equal-weight portfolio
- **Traditional Quant:** ARIMA, GARCH, Fama-French factors
- **Deep Learning:** LSTM, Transformer, pure DRL (PPO, SAC)
- **Neuro-Symbolic (No Vision):** LTN + time series only
- **Vision-Only (No Logic):** ViViT without LTN constraints

Metrics evaluated:

- Sharpe Ratio, Sortino Ratio, Calmar Ratio
- Maximum Drawdown, Value-at-Risk (VaR), Conditional VaR
- Trade execution quality (average slippage, fill rate)
- LTN constraint violation rate (should be ≈ 0)
- Latency percentiles (p50, p95, p99, p99.9)

8 Future Directions: Towards Quant 5.0

8.1 Quantum Computing Integration

Quantum algorithms offer potential advantages for:

- **Portfolio Optimization:** Quadratic Unconstrained Binary Optimization (QUBO) via quantum annealing
- **Option Pricing:** Quantum Monte Carlo with exponential speedup
- **Risk Analysis:** Quantum amplitude estimation for tail risk computation

JANUS is designed to integrate quantum co-processors via:

- Modular risk engine interface (classical or quantum backend)
- Hybrid classical-quantum optimization loops
- Benchmarking quantum advantage on specific subproblems

8.2 Continual Learning and Meta-Learning

Current limitations:

- Models are retrained periodically but not continuously
- Catastrophic forgetting when market regimes shift
- Slow adaptation to novel market conditions

Future enhancements:

- **Elastic Weight Consolidation (EWC):** Protect important weights from updates
- **Meta-Learning (MAML):** Learn initialization that adapts quickly to new regimes
- **Lifelong Learning:** Incremental schema formation without full retraining

8.3 Multi-Agent Cooperation and Competition

Markets are inherently multi-agent. Future JANUS versions may include:

- **Population-Based Training:** Multiple JANUS instances with diverse strategies
- **Cooperative Agents:** Agents share schemas via federated learning
- **Adversarial Agents:** Simulate market makers, informed traders, noise traders
- **Game-Theoretic Equilibria:** Nash equilibrium strategies in multi-agent auctions

8.4 Regulatory AI and Automated Compliance

Symbolic reasoning can be extended to:

- **Automated Regulation Parsing:** Convert SEC rules to LTN predicates
- **Real-Time Compliance Monitoring:** Flag potential violations before execution
- **Regulatory Stress Testing:** Simulate proposed rule changes on strategy performance

9 Conclusion: The Path Forward

Project JANUS represents a synthesis of multiple frontiers in AI and computational finance:

- **Computer Vision:** GAF transforms time series into images, unlocking CNNs and ViTs
- **Symbolic AI:** LTN embeds logical constraints as differentiable predicates
- **Neuroscience:** Neuromorphic architecture mirrors biological brain organization
- **Systems Engineering:** Rust-first implementation ensures safety and performance
- **Memory Systems:** Wake-sleep cycle separates real-time inference from reflective learning

The dual architecture—Forward for real-time trading, Backward for consolidation—mirrors the biological imperative of balancing immediate response with long-term adaptation. By decoupling these concerns, JANUS achieves both microsecond latencies and deep contemplative learning.

The neuromorphic mapping is not metaphorical. Each brain region corresponds to a specific computational challenge in autonomous trading:

- Visual cortex detects patterns in transformed time series
- Prefrontal cortex enforces regulatory and risk constraints
- Hippocampus stores episodic experiences for replay
- Basal ganglia adjudicates action selection via dual pathways
- Cerebellum predicts execution outcomes
- Amygdala implements circuit breakers and risk alerts

This architecture is *explainable by design*. Every decision generates an audit trail tracing:

1. Which visual patterns were detected (Grad-CAM)
2. Which logical constraints were evaluated (LTN trace)
3. Which modalities were trusted (GCA gating)
4. Why the action was authorized (Go/No-Go scores)

This transparency is not incidental—it is the system's core value proposition. In an era of black box AI failures, JANUS offers a "glass box" alternative: a system that is both powerful and accountable.

9.1 Companion Documents

This main document provides the conceptual framework. Technical details are in:

1. **janus_forward.tex** — Algorithms for visual encoding, LTN reasoning, multimodal fusion, and decision-making in the Forward service
2. **janus_backward.tex** — Memory hierarchy, prioritized replay, SWR simulation, UMAP visualization, and vector database integration in the Backward service
3. **janus_neuromorphic_architecture.tex** — Complete neuromorphic mapping, brain region implementations, information flow diagrams, and architectural invariants
4. **janus_rust_implementation.tex** — Rust module structure, ML framework strategy, async service architecture, deployment pipelines, and implementation roadmap

9.2 Call to Action

The transition from Quant 3.0 to Quant 4.0 is not merely an incremental improvement—it is a paradigm shift. The systems we build today will shape the financial markets of the next decade. JANUS offers a blueprint for autonomous trading systems that are:

- **Powerful:** Leveraging state-of-the-art deep learning for pattern recognition
- **Safe:** Enforcing constraints through symbolic reasoning and fail-safe design
- **Transparent:** Generating auditable explanations for every decision
- **Adaptive:** Learning from experience through biologically-inspired memory systems
- **Scalable:** Implemented in Rust for production-grade performance and reliability

The future of quantitative finance will be shaped not by monolithic black boxes, but by hybrid systems that integrate the best of connectionist and symbolic AI, guided by the architectural principles discovered through millions of years of biological evolution.

JANUS is the first step on this path. The journey continues.

"The Roman god Janus is the god of transitions, passages, and new beginnings.
Looking simultaneously to the past and the future,
he embodies the duality required for true intelligence:
learning from history while adapting to the unknown."

References

1. Wang, Z., & Oates, T. (2015). *Imaging time-series to improve classification and imputation*. Proceedings of IJCAI.

2. Arnab, A., Dehghani, M., Heigold, G., Sun, C., Lučić, M., & Schmid, C. (2021). *ViViT: A Video Vision Transformer*. ICCV 2021.

3. Badreddine, S., d'Avila Garcez, A., Serafini, L., & Spranger, M. (2022). *Logic Tensor Networks*. Artificial Intelligence, 303, 103649.

4. Almgren, R., & Chriss, N. (2001). *Optimal execution of portfolio transactions*. Journal of Risk, 3, 5-40.

5. Daw, N. D., Niv, Y., & Dayan, P. (2005). *Uncertainty-based competition between prefrontal and dorsolateral striatal systems for behavioral control*. Nature Neuroscience, 8(12), 1704-1711.

6. Buzsáki, G. (2015). *Hippocampal sharp wave-ripple: A cognitive biomarker for episodic memory and planning*. Hippocampus, 25(10), 1073-1188.

7. McClelland, J. L., McNaughton, B. L., & O'Reilly, R. C. (1995). *Why there are complementary learning systems in the hippocampus and neocortex*. Psychological Review, 102(3), 419.

8. McInnes, L., Healy, J., & Melville, J. (2018). *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. arXiv:1802.03426.

9. Easley, D., López de Prado, M. M., & O'Hara, M. (2012). *Flow toxicity and liquidity in a high-frequency world*. The Review of Financial Studies, 25(5), 1457-1493.

10. Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). *Prioritized experience replay*. ICLR 2016.

11. Veličković, P., Ying, R., Padovano, M., Hadsell, R., & Blundell, C. (2019). *Neural execution of graph algorithms*. ICLR 2020.

12. Ansari, A. F., et al. (2024). *Chronos: Learning the language of time series*. Amazon Science.

13. Argyris, C., & Schön, D. A. (1974). *Theory in practice: Increasing professional effectiveness*. Jossey-Bass.

Appendix: Implementation Checklist

Forward Service (Rust)

- ☐ GAF transformation module with learnable parameters
- ☐ GAF video generation with sliding windows
- ☐ ONNX Runtime integration for ViViT inference
- ☐ LTN predicate evaluation engine
- ☐ Lukasiewicz t-norm implementations
- ☐ Gated cross-attention fusion
- ☐ Basal ganglia dual pathways (Go/No-Go)
- ☐ Cerebellar forward model for slippage prediction
- ☐ Lock-free experience buffer writes
- ☐ gRPC API for order submission
- ☐ Async runtime (Tokio) for market data streams
- ☐ Circuit breaker implementations (amygdala)
- ☐ Audit logging (JSON structured logs)
- ☐ Prometheus metrics export

Backward Service (Rust)

- ☐ Prioritized replay buffer with SumTree
- ☐ SWR simulation with time compression
- ☐ Importance sampling correction
- ☐ Schema formation via clustering
- ☐ Recall-gated consolidation
- ☐ Qdrant client for vector database
- ☐ UMAP projection (ONNX or native)
- ☐ AlignedUMAP for multi-epoch analysis

- ☐ Parametric UMAP for real-time anomaly detection
- ☐ Rayon parallel batch processing
- ☐ Model export and versioning
- ☐ Schema pruning logic

Training Gateway (Python)

- ☐ PyTorch training loop
- ☐ FastAPI REST endpoints
- ☐ Celery task queue for async jobs
- ☐ ONNX export pipeline
- ☐ Model validation scripts
- ☐ Hyperparameter tuning (Optuna)
- ☐ Experiment tracking (MLflow or Weights & Biases)

Infrastructure

- ☐ Docker Compose for local development
- ☐ Kubernetes manifests (Deployments, Services, ConfigMaps)
- ☐ Helm charts for versioned releases
- ☐ PostgreSQL for metadata
- ☐ Qdrant for vector storage
- ☐ Redis for pub/sub and caching
- ☐ Prometheus + Grafana monitoring
- ☐ CI/CD pipeline (GitHub Actions or GitLab CI)
- ☐ Shadow deployment workflow
- ☐ Automated backtesting on commit

Validation

- ☐ Unit tests (>80% coverage)
- ☐ Integration tests (end-to-end flows)
- ☐ Synthetic market simulations
- ☐ Historical backtest suite
- ☐ Black swan stress tests
- ☐ Paper trading validation
- ☐ Latency profiling
- ☐ Reality gap analysis

JANUS FORWARD

Wake State: Logic Trading Algorithm

Real-Time Decision Making, Pattern Recognition, and Trade Execution

Classification: Technical Implementation Guide

Version: 1.0 (Implementation-Ready)

Author: Jordan Smith
github.com/nuniesmith

Date: December 26, 2025

JANUS Forward Overview:

- **Purpose:** Real-time trading decisions during market hours
- **Hot Path:** Low-latency, high-throughput execution
- **Components:** Visual pattern recognition, symbolic reasoning, multimodal fusion, execution control
- **Goal:** Neuro-symbolic trading that combines deep learning with logical constraints

Abstract

JANUS Forward represents the "wake state" of the JANUS trading system, responsible for all real-time decision-making during market hours. This document provides a comprehensive mathematical and implementation specification for the Forward service, which combines:

- **Visual Pattern Recognition** using Gramian Angular Fields (GAF) and Video Vision Transformers (ViViT)
- **Symbolic Reasoning** via Logic Tensor Networks (LTN) for constraint satisfaction
- **Multimodal Fusion** integrating time series, visual, and textual market data
- **Dual-Pathway Decision Making** inspired by basal ganglia architecture

The Forward service operates on a hot path with strict latency requirements, implementing end-to-end gradient flow through differentiable market simulation while maintaining regulatory compliance through symbolic constraints.

Contents

1 Visual Pattern Recognition: DiffGAF and ViViT

The visual subsystem transforms time series data into spatiotemporal images, enabling the system to "see" market patterns that traditional numerical methods miss.

1.1 Mathematical Foundation: Gramian Angular Fields

1.1.1 Input Preprocessing

Given a univariate time series of length N:

$$X = \{x_1, x_2, \ldots, x_N\} \in \mathbb{R}^N \tag{1}$$

1.1.2 Step 1: Learnable Normalization

The time series is normalized using learnable parameters $\alpha \in \mathbb{R}^+$ and $\beta \in \mathbb{R}$:

$$\tilde{x}_i = \tanh\left(\frac{x_i - \min(X)}{\max(X) - \min(X) + \epsilon} \cdot \alpha + \beta\right) \tag{2}$$

where $\epsilon = 10^{-8}$ prevents division by zero. The normalized values $\tilde{x}_i \in [-1, 1]$ are constrained to the domain of the cosine function.

Implementation Note: α and β are learnable parameters initialized as:

$$\alpha_0 = 1.0 \tag{3}$$

$$\beta_0 = 0.0 \tag{4}$$

These are optimized via backpropagation through the entire pipeline.

1.1.3 Step 2: Polar Coordinate Transformation

Each normalized value is mapped to polar coordinates:

$$\begin{cases} \phi_i = \arccos(\tilde{x}_i), & \tilde{x}_i \in [-1, 1] \\ r_i = \frac{i}{N}, & i \in \{1, 2, \ldots, N\} \end{cases} \tag{5}$$

where $\phi_i \in [0, \pi]$ is the angular component and $r_i \in [0, 1]$ is the radial component (normalized time index).

1.1.4 Step 3: Gramian Field Generation

Two Gramian Angular Fields are computed:

Gramian Angular Summation Field (GASF):

$$\text{GASF}_{i,j} = \cos(\phi_i + \phi_j) = \tilde{x}_i \tilde{x}_j - \sqrt{1 - \tilde{x}_i^2}\sqrt{1 - \tilde{x}_j^2} \tag{6}$$

Gramian Angular Difference Field (GADF):

$$\text{GADF}_{i,j} = \sin(\phi_i - \phi_j) = \sqrt{1 - \tilde{x}_i^2}\tilde{x}_j - \tilde{x}_i\sqrt{1 - \tilde{x}_j^2} \tag{7}$$

The result is two $N \times N$ matrices (images) where:

- The main diagonal ($i = j$) contains the original normalized values
- Off-diagonal elements encode temporal correlations
- GASF captures summation relationships
- GADF captures difference relationships

1.1.5 Implementation Algorithm

Algorithm 1 DiffGAF Transformation

Require: Time series $X \in \mathbb{R}^N$, learnable params α, β
Ensure: GASF and GADF matrices $\in \mathbb{R}^{N \times N}$

$X_{\min} \leftarrow \min(X)$, $X_{\max} \leftarrow \max(X)$
$\tilde{X} \leftarrow \tanh\left(\frac{X - X_{\min}}{X_{\max} - X_{\min} + \epsilon} \cdot \alpha + \beta\right)$
$\Phi \leftarrow \arccos(\tilde{X})$ ▷ Element-wise arccos
Initialize GASF $\leftarrow \mathbf{0}_{N \times N}$, GADF $\leftarrow \mathbf{0}_{N \times N}$
for $i = 1$ to N **do**
    for $j = 1$ to N **do**
        $\text{GASF}_{i,j} \leftarrow \cos(\Phi_i + \Phi_j)$
        $\text{GADF}_{i,j} \leftarrow \sin(\Phi_i - \Phi_j)$
    end for
end for
return GASF, GADF

1.2 3D Spatiotemporal Manifolds: GAF Video

1.2.1 Sliding Window GAF Video Generation

Given a time series $X = \{x_1, x_2, \ldots, x_T\}$ of length T, we generate a sequence of overlapping GAF frames:

$$\mathcal{V} = \{GAF(X_{t:t+w}), GAF(X_{t+s:t+w+s}), \ldots, GAF(X_{t+(F-1)s:t+w+(F-1)s})\} \tag{8}$$

where:

- w = window size (e.g., 60 timesteps)
- s = stride (e.g., 10 timesteps)
- F = number of frames (e.g., 16 frames)

Each frame is a $2 \times N \times N$ tensor (GASF + GADF channels). The complete video tensor is:

$$\mathcal{V} \in \mathbb{R}^{F \times 2 \times N \times N} \tag{9}$$

1.2.2 Mathematical Formulation

For frame $f \in \{0, 1, \ldots, F-1\}$:

$$\mathcal{V}_f = \begin{bmatrix} \text{GASF}(X_{t+fs:t+w+fs}) \\ \text{GADF}(X_{t+fs:t+w+fs}) \end{bmatrix} \tag{10}$$

1.3 Video Vision Transformer (ViViT)

1.3.1 Architecture Overview

ViViT processes the 3D tensor $\mathcal{V}$ using factorized spatial-temporal attention.

1.3.2 Patch Embedding

Each frame is divided into patches. For a frame of size $H \times W$ with patch size P:

$$N_p = \frac{H}{P} \times \frac{W}{P} \tag{11}$$

patches per frame.

The patch embedding for patch (i, j) in frame f is:

$$\mathbf{z}_{f,i,j}^{(0)} = \mathbf{E} \cdot \text{flatten}(\mathcal{V}_{f,i:i+P,j:j+P}) + \mathbf{p}_{f,i,j} \tag{12}$$

where:

- $\mathbf{E} \in \mathbb{R}^{d \times (2P^2)}$ is the embedding matrix
- $\mathbf{p}_{f,i,j}$ is the positional encoding (spatial + temporal)

1.3.3 Spatial Attention

Within each frame f, spatial self-attention is computed:

$$\text{Attention}(\mathbf{Q}_s, \mathbf{K}_s, \mathbf{V}_s) = \text{softmax}\left(\frac{\mathbf{Q}_s \mathbf{K}_s^\top}{\sqrt{d_h}}\right)\mathbf{V}_s \tag{13}$$

where $\mathbf{Q}_s, \mathbf{K}_s, \mathbf{V}_s$ are queries, keys, and values from spatial patches.

1.3.4 Temporal Attention

Across frames, temporal attention captures evolution:

$$\text{Attention}(\mathbf{Q}_t, \mathbf{K}_t, \mathbf{V}_t) = \text{softmax}\left(\frac{\mathbf{Q}_t \mathbf{K}_t^\top}{\sqrt{d_h}}\right)\mathbf{V}_t \tag{14}$$

1.3.5 Output Embedding

The final visual embedding vector is:

$$\mathbf{e}_{\text{visual}} = \text{MLP}(\text{GlobalPool}(\mathbf{Z}^{(L)})) \in \mathbb{R}^{d_{\text{embed}}} \tag{15}$$

where L is the number of transformer layers and d_{embed} is the embedding dimension (e.g., 768).

2 Logic Tensor Networks: Symbolic Reasoning Engine

The LTN subsystem ensures that all trading decisions satisfy regulatory and risk management constraints through differentiable first-order logic.

2.1 Mathematical Foundation

2.1.1 Grounding Function

Let $\mathcal{G} : \mathcal{S} \to \mathbb{R}^n$ be a grounding function that maps logical symbols to tensors:

$$\mathcal{G} : \text{Constants} \cup \text{Predicates} \cup \text{Functions} \to \mathbb{R}^n \tag{16}$$

2.1.2 Predicate Grounding

A predicate P with arity k is grounded as a neural network:

$$\mathcal{G}(P) : \mathbb{R}^{k \times d} \to [0, 1] \tag{17}$$

where d is the embedding dimension.

For example, *IsVolatile*$(market)$ is implemented as:

$$\mathcal{G}(\textit{IsVolatile})(\mathbf{e}_{\text{market}}) = \text{sigmoid}(\mathbf{W}_v \mathbf{e}_{\text{market}} + b_v) \tag{18}$$

where $\mathbf{W}_v \in \mathbb{R}^{1\times d}$ and $b_v \in \mathbb{R}$ are learnable parameters.

2.2 Lukasiewicz T-Norm Operations

2.2.1 Conjunction (AND)

$$\mathcal{G}(A \wedge B) = \max(0, \mathcal{G}(A) + \mathcal{G}(B) - 1) \tag{19}$$

2.2.2 Disjunction (OR)

$$\mathcal{G}(A \vee B) = \min(1, \mathcal{G}(A) + \mathcal{G}(B)) \tag{20}$$

2.2.3 Negation (NOT)

$$\mathcal{G}(\neg A) = 1 - \mathcal{G}(A) \tag{21}$$

2.2.4 Implication (IF-THEN)

$$\mathcal{G}(A \to B) = \min(1, 1 - \mathcal{G}(A) + \mathcal{G}(B)) \tag{22}$$

2.2.5 Universal Quantification (FOR ALL)

For a formula $\phi(x)$ with free variable x:

$$\mathcal{G}(\forall x : \phi(x)) = \min_{x\in\mathcal{D}} \mathcal{G}(\phi(x)) \tag{23}$$

where $\mathcal{D}$ is the domain of x.

2.2.6 Existential Quantification (EXISTS)

$$\mathcal{G}(\exists x : \phi(x)) = \max_{x\in\mathcal{D}} \mathcal{G}(\phi(x)) \tag{24}$$

2.3 Knowledge Base Formulation

2.3.1 Wash Sale Constraint

The wash sale rule is encoded as:

$$\forall t, \forall k \in [1, 30] : \neg(\textit{SaleAtLoss}(t) \wedge \textit{Buy}(t + k)) \tag{25}$$

In grounded form:

$$\mathcal{G}(\text{WashSale}) = \min_{t,k\in[1,30]} [1 - \max(0, \mathcal{G}(\textit{SaleAtLoss})(t) + \mathcal{G}(\textit{Buy})(t+k) - 1)] \tag{26}$$

2.3.2 Almgren-Chriss Risk Constraint

$$\forall v : \textit{Volatile}(Market) \rightarrow \textit{Impact}(v) < \textit{Threshold}(\sigma) \tag{27}$$

Grounded:

$$\mathcal{G}(\text{ACRisk}) = \min_{v} [\min(1, 1 - \mathcal{G}(\textit{Volatile}) + \mathcal{G}(\textit{Impact}(v) < \textit{Threshold}(\sigma)))] \tag{28}$$

where:

$$\textit{Threshold}(\sigma) = \eta \cdot \sigma \cdot \sqrt{\frac{v}{V}} \tag{29}$$

with η = impact coefficient, σ = volatility, v = trade size, V = average volume.

2.3.3 VPIN Toxicity Constraint

$$\forall t : \textit{VPIN}_t > \tau_{\text{VPIN}} \rightarrow \textit{HaltTrading}(t) \tag{30}$$

Grounded:

$$\mathcal{G}(\text{VPIN}) = \min_{t} [\min(1, 1 - \mathcal{G}(\textit{VPIN}_t > \tau_{\text{VPIN}}) + \mathcal{G}(\textit{HaltTrading})(t))] \tag{31}$$

2.4 Logical Loss Function

2.4.1 Satisfiability Aggregation

For a knowledge base $\mathcal{K} = \{\phi_1, \phi_2, \ldots, \phi_m\}$:

$$\text{SatAgg}(\mathcal{K}) = \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{G}(\phi_i)^p \right)^{1/p} \tag{32}$$

where p is the generalized mean parameter (typically $p = 2$ for quadratic mean).

2.4.2 Logical Loss

$$\mathcal{L}_{\text{logic}}(\theta) = 1 - \text{SatAgg}(\mathcal{K}) \tag{33}$$

2.4.3 Combined Loss

The total loss combines predictive and logical components:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{predictive}} + \lambda_{\text{logic}} \cdot \mathcal{L}_{\text{logic}} \tag{34}$$

where λ_{logic} is a hyperparameter (typically 0.1 to 1.0).

3 Multimodal Fusion: Gated Cross-Attention

The fusion subsystem integrates visual, temporal, and textual market information into a unified representation.

3.1 Input Modalities

The system receives three input streams:

$$\mathbf{H}_{\text{TS}} \in \mathbb{R}^{L_{\text{TS}} \times d} \quad \text{(Time Series Tokens from Chronos-Bolt)} \tag{35}$$

$$\mathbf{H}_{\text{Vis}} \in \mathbb{R}^{L_{\text{Vis}} \times d} \quad \text{(Visual Embeddings from ViViT)} \tag{36}$$

$$\mathbf{H}_{\text{Text}} \in \mathbb{R}^{L_{\text{Text}} \times d} \quad \text{(Text Embeddings from FinBERT)} \tag{37}$$

3.2 Gated Cross-Attention Mechanism

3.2.1 Attention Computation

For primary modality m and auxiliary modality n:

$$\alpha_{m \to n} = \text{softmax}\left(\frac{\mathbf{Q}_m \mathbf{K}_n^\top}{\sqrt{d_h}} \right) \tag{38}$$

where:

$$\mathbf{Q}_m = \mathbf{H}_m \mathbf{W}_Q \tag{39}$$

$$\mathbf{K}_n = \mathbf{H}_n \mathbf{W}_K \tag{40}$$

$$\mathbf{V}_n = \mathbf{H}_n \mathbf{W}_V \tag{41}$$

3.2.2 Gating Mechanism

The gating scalar is computed as:

$$\lambda_{\text{gate}} = \text{sigmoid}(\mathbf{W}_g [\mathbf{H}_m; \mathbf{H}_n] + b_g) \tag{42}$$

where $[\cdot;\cdot]$ denotes concatenation.

3.2.3 Fused Representation

$$\mathbf{H}_{\text{fused}} = \mathbf{H}_m + \lambda_{\text{gate}} \cdot \alpha_{m \to n} \mathbf{V}_n \tag{43}$$

3.3 Multi-Modal Fusion Pipeline

The complete fusion process:

$$\mathbf{H}_1 = \mathbf{H}_{\text{TS}} + \lambda_{\text{vis}} \cdot \text{CrossAttn}(\mathbf{H}_{\text{TS}}, \mathbf{H}_{\text{Vis}}) \tag{44}$$

$$\mathbf{H}_2 = \mathbf{H}_1 + \lambda_{\text{text}} \cdot \text{CrossAttn}(\mathbf{H}_1, \mathbf{H}_{\text{Text}}) \tag{45}$$

$$\mathbf{e}_{\text{final}} = \text{GlobalPool}(\mathbf{H}_2) \tag{46}$$

4 Decision Engine: Basal Ganglia Pathways

The decision engine implements a dual-pathway architecture inspired by the basal ganglia, with separate "go" and "no-go" pathways.

4.1 Praxeological Motor: Dual Pathways

4.1.1 Direct Pathway (Go Signal)

The direct pathway generates action proposals:

$$a_{\text{proposed}} = \arg\max_{a \in \mathcal{A}} \left[\mathbf{W}_{\text{alpha}} \mathbf{e}_{\text{final}} + b_{\text{alpha}} \right]_a \tag{47}$$

where $\mathcal{A}$ is the action space (BUY, SELL, HOLD, or continuous trade sizes).

4.1.2 Indirect Pathway (No-Go Signal)

The indirect pathway computes risk veto:

$$v_{\text{risk}} = \text{sigmoid}(\mathbf{W}_{\text{risk}} [\mathbf{e}_{\text{final}}; \text{VPIN}; \sigma_{\text{market}}] + b_{\text{risk}}) \tag{48}$$

4.1.3 Final Action

The final action is gated by the risk veto:

$$a_{\text{final}} = \begin{cases} a_{\text{proposed}} & \text{if } v_{\text{risk}} < \tau_{\text{risk}} \text{ AND } \mathcal{L}_{\text{logic}} < \tau_{\text{logic}} \\ \text{HOLD} & \text{otherwise} \end{cases} \tag{49}$$

where τ_{risk} is the risk threshold (e.g., 0.7) and τ_{logic} is the logical constraint threshold (e.g., 0.1).

4.2 Cerebellar Forward Model

4.2.1 Market Impact Prediction

The forward model predicts execution price:

$$\hat{p}_{\text{exec}} = f_{\text{forward}}(\mathbf{s}_{\text{LOB}}, v, a_{\text{final}}) \tag{50}$$

where $\mathbf{s}_{\text{LOB}}$ is the limit order book state.

4.2.2 Sensory Prediction Error

$$\text{SPE} = |p_{\text{actual}} - \hat{p}_{\text{exec}}| \tag{51}$$

4.2.3 Trajectory Adjustment

The execution trajectory is adjusted:

$$v_{\text{adjusted}} = v_{\text{original}} - \eta_{\text{cerebellar}} \cdot \text{SPE} \cdot \nabla_v \text{SPE} \tag{52}$$

5 Implementation Checklist

sec:checklist

This section provides a sequential checklist for implementing JANUS Forward.

5.1 Core Components

1. **Visual Pattern Recognition Module**
   - ☐ Implement DiffGAF normalization with learnable α, β
   - ☐ Implement polar coordinate transformation
   - ☐ Implement GASF and GADF computation
   - ☐ Implement sliding window GAF video generation
   - ☐ Implement ViViT patch embedding
   - ☐ Implement spatial attention mechanism
   - ☐ Implement temporal attention mechanism
   - ☐ Test on sample time series data
   - ☐ Benchmark latency (target: <50ms for inference)
2. **Logic Tensor Networks Module**
   - ☐ Implement grounding function framework
   - ☐ Implement predicate neural networks
   - ☐ Implement Lukasiewicz T-norm operations
   - ☐ Implement universal/existential quantification
   - ☐ Encode wash sale constraint
   - ☐ Encode Almgren-Chriss constraint
   - ☐ Encode VPIN constraint
   - ☐ Implement satisfiability aggregation
   - ☐ Implement logical loss function
   - ☐ Test constraint satisfaction with edge cases
   - ☐ Benchmark latency (target: <10ms for evaluation)
3. **Multimodal Fusion Module**
   - ☐ Implement time series tokenization (Chronos-Bolt integration)
   - ☐ Implement visual embedding extraction

- ☐ Implement text embedding (FinBERT integration)
- ☐ Implement gated cross-attention mechanism
- ☐ Implement multi-modal fusion pipeline
- ☐ Test fusion on sample multimodal data
- ☐ Validate attention weight distributions

4. **Decision Engine Module**

- ☐ Implement direct pathway (alpha motor)
- ☐ Implement indirect pathway (risk motor)
- ☐ Implement risk-gated action selection
- ☐ Implement logic-gated action selection
- ☐ Implement cerebellar forward model
- ☐ Implement sensory prediction error computation
- ☐ Implement trajectory adjustment
- ☐ Test end-to-end decision making
- ☐ Validate constraint adherence in production scenarios

5.2 Integration & Testing

1. **End-to-End Pipeline**

- ☐ Connect all modules into unified forward pass
- ☐ Implement gradient flow verification
- ☐ Test backpropagation through entire pipeline
- ☐ Validate differentiable constraint satisfaction

2. **Performance Optimization**

- ☐ Profile latency bottlenecks
- ☐ Optimize tensor operations for GPU
- ☐ Implement model quantization (INT8/FP16)
- ☐ Add batching support for parallel inference
- ☐ Target: <100ms end-to-end latency

3. **Safety & Validation**

- ☐ Add input validation and sanitization

- ☐ Implement kill switch integration
- ☐ Add logging for all trading decisions
- ☐ Implement emergency halt on constraint violation
- ☐ Test failure modes (network outage, invalid data, etc.)

5.3 Deployment Readiness

1. **Production Hardening**
   - ☐ Remove all panic!() calls
   - ☐ Replace unwrap() with proper error handling
   - ☐ Add comprehensive error types
   - ☐ Implement graceful degradation
   - ☐ Add health check endpoints
2. **Monitoring & Observability**
   - ☐ Add metrics export (Prometheus format)
   - ☐ Implement distributed tracing
   - ☐ Log all constraint violations
   - ☐ Monitor inference latency
   - ☐ Track prediction accuracy

6 Rust Implementation Considerations

sec:rust

6.1 Hot Path Optimization

The Forward service must maintain low latency (<100ms end-to-end). Key Rust optimizations:

- **Zero-copy operations:** Use `ndarray` views instead of clones
- **SIMD acceleration:** Leverage `packed_simd` for GAF computation
- **Async runtime:** Use `tokio` for non-blocking I/O
- **Memory pooling:** Pre-allocate tensor buffers to avoid allocation overhead

6.2 ML Framework Integration

6.2.1 Option 1: PyTorch via tch-rs

- Pros: Full PyTorch ecosystem, easy model export
- Cons: Requires LibTorch, larger binary size

6.2.2 Option 2: ONNX Runtime via ort

- Pros: Lightweight, cross-platform, optimized inference
- Cons: Limited to inference, requires model conversion
- **Recommended for production**

6.2.3 Option 3: Candle (Hugging Face)

- Pros: Pure Rust, no C++ dependencies
- Cons: Younger ecosystem, fewer pre-trained models
- **Recommended for future migration**

6.3 Error Handling Strategy

```rust
// Custom error types for Forward service
#[derive(Debug, thiserror::Error)]
pub enum ForwardError {
    #[error("GAF transformation failed: {0}")]
    GafError(String),

    #[error("LTN constraint violation: {constraint}")]
    ConstraintViolation { constraint: String },

    #[error("Model inference failed: {0}")]
    InferenceError(String),

    #[error("Risk threshold exceeded: {risk_score}")]
    RiskVeto { risk_score: f64 },
}

// Result type alias
pub type ForwardResult<T> = Result<T, ForwardError>;
```

References

[1] Jordan Smith, "Project JANUS: Implementation Guide v1.0," 2025.

[2] Wang, Oates, "Encoding Time Series as Images for Visual Inspection and Classification Using Tiled Convolutional Neural Networks," AAAI 2015.

[3] Arnab et al., "ViViT: A Video Vision Transformer," ICCV 2021.

[4] Badreddine et al., "Logic Tensor Networks," Artificial Intelligence, 2022.

[5] Ansari et al., "Chronos: Learning the Language of Time Series," arXiv:2403.07815, 2024.

[6] Araci, "FinBERT: Financial Sentiment Analysis with Pre-trained Language Models," arXiv:1908.10063, 2019.

[7] Almgren, Chriss, "Optimal Execution of Portfolio Transactions," Journal of Risk, 2000.

[8] Easley et al., "Flow Toxicity and Liquidity in a High-frequency World," Review of Financial Studies, 2012.

JANUS BACKWARD

Sleep State: Memory Management

Knowledge Consolidation, Schema Formation, and Long-Term Learning

Classification: Technical Implementation Guide

Version: 1.0 (Implementation-Ready)

Author: Jordan Smith
github.com/nuniesmith

Date: December 26, 2025

JANUS Backward Overview:

- **Purpose:** Offline memory consolidation and schema learning
- **Cold Path:** Batch processing during market closure
- **Components:** Three-timescale memory, prioritized replay, UMAP visualization
- **Goal:** Transform raw experiences into abstract knowledge structures

Abstract

JANUS Backward represents the "sleep state" of the JANUS trading system, responsible for offline memory consolidation, schema formation, and long-term learning during market closure. This document provides a comprehensive mathematical and implementation specification for the Backward service, which implements:

- **Three-Timescale Memory Architecture** spanning short-term (hippocampus), medium-term (SWR replay), and long-term (neocortex) storage
- **Prioritized Experience Replay** using TD-error, logical violation scores, and reward magnitude
- **Sharp Wave Ripple Simulation** for time-compressed memory consolidation
- **Recall-Gated Learning** that filters updates based on familiarity and logical validity
- **UMAP Visualization** for real-time cognitive monitoring and anomaly detection

The Backward service operates on a cold path with no strict latency requirements, enabling sophisticated batch processing and offline optimization that would be infeasible during live trading.

Contents

1 Memory Hierarchy: Three-Timescale Architecture

The memory system is organized into three distinct timescales, each with specialized functions and computational properties.

1.1 Short-Term Memory (Hippocampus)

The hippocampal subsystem provides rapid encoding of recent experiences with pattern separation to prevent interference.

1.1.1 Episodic Buffer

The hippocampus maintains an episodic buffer of recent transitions:

$$\mathcal{B}_{\text{STM}} = \{(\mathbf{s}_t, a_t, r_t, \mathbf{s}_{t+1})\}_{t=1}^{T_{\text{episode}}} \tag{1}$$

where each tuple represents a state-action-reward-nextstate transition.

Implementation Details:

- Maximum capacity: $|\mathcal{B}_{\text{STM}}| \leq 10,000$ transitions
- FIFO replacement policy when capacity exceeded
- Indexed by timestamp for temporal queries

1.1.2 Pattern Separation

To prevent catastrophic interference between similar market states, the hippocampus implements pattern separation:

$$\mathbf{h}_{\text{separated}} = \text{ReLU}(\mathbf{W}_{\text{sep}}\mathbf{s} + \mathbf{b}_{\text{sep}}) \tag{2}$$

where $\mathbf{W}_{\text{sep}} \in \mathbb{R}^{d_h \times d_s}$ is initialized to promote orthogonality.

Orthogonality Initialization:

$$\mathbf{W}_{\text{sep}} \sim \mathcal{N}(0, \sigma^2), \quad \text{where } \sigma = \sqrt{\frac{2}{d_s + d_h}} \tag{3}$$

During training, add orthogonality regularization:

$$\mathcal{L}_{\text{ortho}} = \lambda_{\text{ortho}} \cdot ||\mathbf{W}_{\text{sep}}^\top \mathbf{W}_{\text{sep}} - \mathbf{I}||_F^2 \tag{4}$$

1.1.3 Sparse Encoding

The hippocampus uses sparse representations to maximize information capacity:

$$\mathbf{c}_{\text{sparse}} = \text{TopK}(\mathbf{h}_{\text{separated}}, k) \tag{5}$$

where TopK selects the k largest activations and zeros others.

Sparsity Level:

$$k = \lceil \rho \cdot d_h \rceil, \quad \rho \in [0.05, 0.15] \tag{6}$$

Typically $\rho = 0.1$ (10% activation).

1.2 Medium-Term Consolidation (SWR Simulator)

The Sharp Wave Ripple (SWR) simulator implements prioritized replay with time compression, mimicking biological memory consolidation during sleep.

1.2.1 Replay Prioritization

Each transition is assigned a priority score combining three components:

$$p_i = |\delta_i| + \lambda_{\text{logic}} \cdot v_i + \lambda_{\text{reward}} \cdot |r_i| \tag{7}$$

where:

- δ_i = TD-error: $r_i + \gamma Q(\mathbf{s}_{i+1}, a_{i+1}) - Q(\mathbf{s}_i, a_i)$
- v_i = logical violation score from LTN (higher = more constraint violations)
- r_i = reward magnitude (prioritize high-reward experiences)
- $\lambda_{\text{logic}} = 2.0$ (weight for constraint violations)
- $\lambda_{\text{reward}} = 0.5$ (weight for reward magnitude)

Rationale:

- High TD-error $\rightarrow$ surprising transitions that require learning
- High violation score $\rightarrow$ dangerous patterns to avoid
- High reward $\rightarrow$ successful strategies to reinforce

1.2.2 Sampling Probability

Transitions are sampled stochastically with probability proportional to priority:

$$P(i) = \frac{p_i^\alpha}{\sum_{j=1}^{|\mathcal{B}_{\text{STM}}|} p_j^\alpha} \tag{8}$$

where $\alpha \in [0, 1]$ controls prioritization strength:

- $\alpha = 0 \rightarrow$ uniform sampling
- $\alpha = 1 \rightarrow$ greedy prioritization
- $\alpha = 0.6 \rightarrow$ recommended default (balanced)

1.2.3 Importance Sampling Correction

To correct for sampling bias, apply importance-sampling weights:

$$w_i = \left(\frac{1}{|\mathcal{B}_{\text{STM}}|} \cdot \frac{1}{P(i)} \right)^\beta \tag{9}$$

where $\beta \in [0, 1]$ is annealed from 0.4 to 1.0 during training.

Normalized weights:

$$\bar{w}_i = \frac{w_i}{\max_j w_j} \tag{10}$$

Gradients are scaled by importance weights:

$$\nabla_\theta \mathcal{L}(\tau_i) \leftarrow \bar{w}_i \cdot \nabla_\theta \mathcal{L}(\tau_i) \tag{11}$$

1.2.4 Time Compression

During replay, transitions are replayed at $C\times$ speed to accelerate consolidation:

$$\Delta t_{\text{replay}} = \frac{\Delta t_{\text{original}}}{C} \tag{12}$$

where $C \in [10, 20]$ is the compression factor (typically $C = 15$).

Biological Motivation: Real hippocampal replay occurs at 10-20$\times$ speed during sleep.

1.2.5 SWR Replay Algorithm

Algorithm 1 Sharp Wave Ripple Replay

Require: Buffer $\mathcal{B}_{\text{STM}}$, compression factor C, batch size B, prioritization exponent α

Ensure: Replay batch $\mathcal{B}_{\text{replay}}$, importance weights $\mathbf{w}$

Compute TD-errors δ_i for all transitions

Compute logical violations v_i via LTN evaluation

Compute priorities $p_i = |\delta_i| + \lambda_{\text{logic}} \cdot v_i + \lambda_{\text{reward}} \cdot |r_i|$

Compute sampling probabilities $P(i) = p_i^\alpha / \sum_j p_j^\alpha$

Sample B transition indices with probabilities $P(i)$

Compute importance weights $w_i = (1/(|\mathcal{B}_{\text{STM}}| \cdot P(i)))^\beta$

Normalize weights $\bar{w}_i = w_i / \max_j w_j$

for each sampled transition $\tau_i = (\mathbf{s}_t, a_t, r_t, \mathbf{s}_{t+1})$ **do**

    Compress time: $\Delta t \leftarrow \Delta t / C$

    Add $(\tau_i, \bar{w}_i)$ to $\mathcal{B}_{\text{replay}}$

end for

return $\mathcal{B}_{\text{replay}}, \mathbf{w}$

1.3 Long-Term Memory (Neocortex)

The neocortical subsystem maintains abstract schemas—statistical summaries of recurring market patterns.

1.3.1 Schema Representation

Each schema k is represented as a Gaussian distribution:

$$\mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{13}$$

where:

$$\boldsymbol{\mu}_k = \frac{1}{|\mathcal{S}_k|} \sum_{\mathbf{s} \in \mathcal{S}_k} \mathbf{s} \tag{14}$$

$$\boldsymbol{\Sigma}_k = \frac{1}{|\mathcal{S}_k|} \sum_{\mathbf{s} \in \mathcal{S}_k} (\mathbf{s} - \boldsymbol{\mu}_k)(\mathbf{s} - \boldsymbol{\mu}_k)^\top \tag{15}$$

$\mathcal{S}_k$ is the set of all states assigned to schema k.

1.3.2 Schema Assignment

New states are assigned to schemas via maximum likelihood:

$$k^* = \arg\max_k \mathcal{N}(\mathbf{s}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{16}$$

If $\max_k \mathcal{N}(\mathbf{s}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) < \tau_{\text{schema}}$, create a new schema.

1.3.3 Recall-Gated Consolidation

Updates to long-term memory are gated by two factors: recall strength (familiarity) and logical validity.

Recall Strength:

$$g(r_{\text{STM}}(\tau)) = \text{sigmoid}(\mathbf{W}_r \mathbf{r}_{\text{STM}} + b_r) \tag{17}$$

where $\mathbf{r}_{\text{STM}}$ is the hippocampal representation of transition τ.

Logical Validity:

$$g_{\text{sym}}(\tau) = \text{SatAgg}(\mathcal{K}_{\text{episode}}) \tag{18}$$

where $\mathcal{K}_{\text{episode}}$ is the knowledge base evaluated on the episode containing τ.

Gated Update Rule:

$$\Delta \mathbf{W}_{\text{LTM}} = \eta_{\text{sleep}} \cdot g(r_{\text{STM}}(\tau)) \cdot g_{\text{sym}}(\tau) \cdot \nabla_{\mathbf{W}} \mathcal{L}_{\text{policy}}(\tau) \tag{19}$$

Only update if both gates exceed thresholds:

$$\text{Update if: } g(r_{\text{STM}}) > \tau_{\text{recall}} \text{ AND } g_{\text{sym}} > \tau_{\text{logic}} \tag{20}$$

Typical thresholds: $\tau_{\text{recall}} = 0.3$, $\tau_{\text{logic}} = 0.7$.

1.3.4 Consolidation Update Rule

For schema k, update mean and covariance:

$$\boldsymbol{\mu}_k^{(t+1)} = (1 - \eta_{\text{schema}})\boldsymbol{\mu}_k^{(t)} + \eta_{\text{schema}} \cdot \mathbf{s}_{\text{new}} \tag{21}$$

$$\boldsymbol{\Sigma}_k^{(t+1)} = (1 - \eta_{\text{schema}})\boldsymbol{\Sigma}_k^{(t)} + \eta_{\text{schema}} \cdot (\mathbf{s}_{\text{new}} - \boldsymbol{\mu}_k^{(t)})(\mathbf{s}_{\text{new}} - \boldsymbol{\mu}_k^{(t)})^\top \tag{22}$$

where η_{schema} is the schema learning rate (typically 0.01).

2 UMAP Visualization: Cognitive Dashboard

UMAP (Uniform Manifold Approximation and Projection) provides a real-time 3D visualization of the system's internal knowledge structure.

2.1 AlignedUMAP for Schema Formation

AlignedUMAP ensures consistency across multiple sleep cycles, enabling tracking of schema evolution over time.

2.1.1 Objective Function

AlignedUMAP minimizes:

$$\mathcal{L}_{\text{AlignedUMAP}} = \sum_{t=1}^{T_{\text{cycles}}} \left[\mathcal{L}_{\text{UMAP}}(\mathbf{X}_t) + \lambda_{\text{align}} \sum_{i,j} w_{ij} ||\mathbf{y}_i^{(t)} - \mathbf{y}_j^{(t-1)}||^2 \right] \tag{23}$$

where:

- $\mathbf{X}_t \in \mathbb{R}^{N_t \times d}$ = high-dimensional embeddings at sleep cycle t
- $\mathbf{y}_i^{(t)} \in \mathbb{R}^3$ = 3D projection of point i at cycle t
- w_{ij} = alignment weights (higher for points in same schema)
- $\lambda_{\text{align}} = 0.1$ = alignment strength

Standard UMAP Loss:

$$\mathcal{L}_{\text{UMAP}}(\mathbf{X}) = \sum_{i,j} \left[v_{ij} \log \frac{v_{ij}}{w_{ij}} + (1 - v_{ij}) \log \frac{1 - v_{ij}}{1 - w_{ij}} \right] \tag{24}$$

where v_{ij} is high-dimensional similarity and w_{ij} is low-dimensional similarity.

2.1.2 Alignment Weights

$$w_{ij} = \begin{cases} 1.0 & \text{if schema}(i) = \text{schema}(j) \\ 0.1 & \text{otherwise} \end{cases} \tag{25}$$

2.1.3 Schema Cluster Detection

Schemas are identified as dense clusters in UMAP space using DBSCAN:

$$\text{Schema}_k = \{\mathbf{y}_i : ||\mathbf{y}_i - \boldsymbol{\mu}_k|| < \tau_{\text{cluster}}\} \tag{26}$$

where τ_{cluster} is the cluster radius (typically 0.5 in normalized UMAP space).

2.2 Parametric UMAP for Real-Time Monitoring

Parametric UMAP learns a neural network mapping for fast projection of new points during live trading.

2.2.1 Neural Network Projection

A feedforward network $f_{\text{UMAP}} : \mathbb{R}^d \to \mathbb{R}^3$ is trained to approximate UMAP projection:

$$\mathbf{y} = f_{\text{UMAP}}(\mathbf{e}; \theta_{\text{UMAP}}) \tag{27}$$

Architecture:

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{W}_1\mathbf{e} + \mathbf{b}_1), \quad \mathbf{h}_1 \in \mathbb{R}^{256} \tag{28}$$

$$\mathbf{h}_2 = \text{ReLU}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2), \quad \mathbf{h}_2 \in \mathbb{R}^{128} \tag{29}$$

$$\mathbf{y} = \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3, \quad \mathbf{y} \in \mathbb{R}^3 \tag{30}$$

Training Objective:

$$\mathcal{L}_{\text{ParametricUMAP}} = \sum_i ||\mathbf{y}_i - f_{\text{UMAP}}(\mathbf{e}_i)||^2 + \mathcal{L}_{\text{UMAP}} \tag{31}$$

2.2.2 Anomaly Detection

During live trading, a point is flagged as anomalous if it falls outside all known schemas:

$$\text{Anomaly}(\mathbf{y}) = \mathbb{1}\left[\min_k ||\mathbf{y} - \boldsymbol{\mu}_k|| > \tau_{\text{anomaly}}\right] \tag{32}$$

where $\tau_{\text{anomaly}} = 2.0$ (units in UMAP space).

Response to Anomalies:

- Log anomaly with full context
- Increase risk threshold temporarily
- Alert human operator if anomaly persists
- Add to high-priority replay buffer

3 Integration with Vector Database (Qdrant)

Long-term memory schemas are persisted in Qdrant for efficient similarity search and retrieval.

3.1 Schema Storage

Each schema is stored as a point in Qdrant:

- **Vector:** $\boldsymbol{\mu}_k \in \mathbb{R}^d$ (schema centroid)

- **Payload:**
  - `schema_id`: Unique identifier
  - `covariance`: Flattened Σ_k
  - `num_points`: $|\mathcal{S}_k|$
  - `avg_reward`: Mean reward for transitions in schema
  - `created_at`: Timestamp
  - `last_updated`: Timestamp

3.2 Similarity Search

Given a new state $\mathbf{s}_{\text{new}}$, retrieve top-k similar schemas:

$$\text{TopK}(\mathbf{s}_{\text{new}}) = \underset{k,|K|=k}{\arg\max} \{\text{cosine}(\mathbf{s}_{\text{new}}, \boldsymbol{\mu}_i)\}_{i=1}^{N_{\text{schemas}}} \tag{33}$$

Use Cases:

- Retrieve historical context during decision-making
- Find similar market conditions for transfer learning
- Identify schema membership for new states

3.3 Periodic Schema Pruning

Remove low-quality schemas to prevent memory bloat:

$$\text{Prune if: } |\mathcal{S}_k| < \tau_{\text{min\_points}} \text{ OR age}(k) > \tau_{\text{max\_age}} \tag{34}$$

where $\tau_{\text{min\_points}} = 10$ and $\tau_{\text{max\_age}} = 90$ days.

4 Sleep Cycle: Complete Algorithm

The sleep cycle runs nightly (or after market close) to consolidate the day's experiences.

Algorithm 2 JANUS Backward Sleep Cycle

Require: Short-term buffer $\mathcal{B}_{\text{STM}}$, long-term schemas $\{\mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\}_k$

Ensure: Updated schemas, trained policy

Phase 1: Prioritized Replay (SWR Simulation)
for n_{replays} iterations (e.g., 1000) **do**
  Sample batch $\mathcal{B}_{\text{replay}}$ using SWR algorithm
  Compute losses: $\mathcal{L}_{\text{policy}}, \mathcal{L}_{\text{logic}}$
  Update policy: $\theta \leftarrow \theta - \eta \cdot \bar{w}_i \cdot \nabla_\theta \mathcal{L}_{\text{total}}$
  Update priorities: $p_i \leftarrow |\delta_i| + \lambda_{\text{logic}} v_i + \lambda_{\text{reward}} |r_i|$
end for

Phase 2: Schema Consolidation
for each transition $\tau_i \in \mathcal{B}_{\text{STM}}$ **do**
  Compute recall gate: $g_{\text{recall}} = \text{sigmoid}(\mathbf{W}_r \mathbf{r}_{\text{STM}} + b_r)$
  Compute logic gate: $g_{\text{logic}} = \text{SatAgg}(\mathcal{K})$
  if $g_{\text{recall}} > \tau_{\text{recall}}$ AND $g_{\text{logic}} > \tau_{\text{logic}}$ **then**
    Find matching schema: $k^* = \arg\max_k \mathcal{N}(\mathbf{s}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$
    if $\mathcal{N}(\mathbf{s}_i; \boldsymbol{\mu}_{k^*}, \boldsymbol{\Sigma}_{k^*}) < \tau_{\text{schema}}$ **then**
      Create new schema: $\boldsymbol{\mu}_{\text{new}} \leftarrow \mathbf{s}_i$, $\boldsymbol{\Sigma}_{\text{new}} \leftarrow \epsilon \mathbf{I}$
    else
      Update schema k^* using consolidation rule
    end if
  end if
end for

Phase 3: UMAP Update
Extract all schema centroids: $\{\boldsymbol{\mu}_k\}_k$
Fit AlignedUMAP with previous cycle alignment
Update parametric UMAP network
Detect new clusters via DBSCAN

Phase 4: Vector Database Sync
Upsert updated schemas to Qdrant
Prune low-quality schemas
Create snapshot for recovery

Phase 5: Metrics & Logging
Compute and log:

- Number of schemas: N_{schemas}
- Mean schema size: $\mathbb{E}[|\mathcal{S}_k|]$
- Constraint satisfaction rate: $\mathbb{E}[\text{SatAgg}(\mathcal{K})]$
- Average TD-error improvement

5 Implementation Checklist

sec:checklist

This section provides a sequential checklist for implementing JANUS Backward.

5.1 Core Components

1. **Short-Term Memory (Hippocampus)**
   - ☐ Implement episodic buffer with FIFO eviction
   - ☐ Implement pattern separation layer
   - ☐ Add orthogonality regularization
   - ☐ Implement TopK sparse encoding
   - ☐ Add timestamp indexing for temporal queries
   - ☐ Test buffer operations (insert, retrieve, evict)
2. **Sharp Wave Ripple (SWR) Simulator**
   - ☐ Implement TD-error computation
   - ☐ Implement logical violation scoring via LTN
   - ☐ Implement composite priority function
   - ☐ Implement prioritized sampling with importance weights
   - ☐ Add time compression simulation
   - ☐ Test replay batch generation
   - ☐ Validate importance weight correction
3. **Long-Term Memory (Neocortex)**
   - ☐ Implement schema representation (Gaussian)
   - ☐ Implement schema assignment via maximum likelihood
   - ☐ Implement recall gate computation
   - ☐ Implement logical validity gate
   - ☐ Implement gated consolidation update
   - ☐ Add schema creation logic
   - ☐ Test schema updates with edge cases
4. **UMAP Visualization**

- ☐ Implement AlignedUMAP objective
- ☐ Add alignment weight computation
- ☐ Implement parametric UMAP network
- ☐ Implement DBSCAN cluster detection
- ☐ Add anomaly detection logic
- ☐ Test visualization updates across cycles
- ☐ Validate cluster stability

5.2 Integration & Storage

1. **Vector Database Integration (Qdrant)**
   - ☐ Set up Qdrant connection
   - ☐ Define schema collection structure
   - ☐ Implement schema upsert operations
   - ☐ Implement similarity search queries
   - ☐ Add periodic pruning logic
   - ☐ Implement backup/restore functionality
   - ☐ Test concurrent access patterns
2. **Sleep Cycle Orchestration**
   - ☐ Implement 5-phase sleep cycle algorithm
   - ☐ Add progress tracking and logging
   - ☐ Implement graceful shutdown on errors
   - ☐ Add checkpoint/resume capability
   - ☐ Test full sleep cycle end-to-end
   - ☐ Validate schema evolution over cycles

5.3 Monitoring & Debugging

1. **Metrics Collection**
   - ☐ Track number of schemas over time
   - ☐ Monitor mean schema size
   - ☐ Track constraint satisfaction rates

- ☐ Monitor TD-error distribution
- ☐ Track UMAP cluster count
- ☐ Log replay batch statistics

2. **Visualization & Debugging**
   - ☐ Export UMAP projections for visualization
   - ☐ Add schema evolution timeline
   - ☐ Visualize priority distributions
   - ☐ Plot constraint satisfaction heatmaps
   - ☐ Add interactive schema browser

5.4 Performance Optimization

1. **Batch Processing**
   - ☐ Parallelize TD-error computation
   - ☐ Vectorize schema likelihood calculations
   - ☐ Batch Qdrant upsert operations
   - ☐ Use GPU for UMAP fitting (if available)
   - ☐ Profile and optimize bottlenecks
   - ☐ Target: <10 minutes for 10k transitions

6 Rust Implementation Considerations

sec:rust

6.1 Cold Path Optimization

Unlike Forward, Backward has no strict latency requirements, allowing focus on throughput and correctness.

- **Batch parallelism:** Use `rayon` for parallel replay processing
- **Memory efficiency:** Use `ndarray` for linear algebra operations
- **Async I/O:** Use `tokio` for non-blocking Qdrant operations
- **Checkpointing:** Serialize intermediate state with `serde`

6.2 Data Structures

6.2.1 Episodic Buffer

```rust
use std::collections::VecDeque;

#[derive(Clone, Debug)]
pub struct Transition {
    pub state: Array1<f32>,
    pub action: Action,
    pub reward: f32,
    pub next_state: Array1<f32>,
    pub timestamp: u64,
}

pub struct EpisodicBuffer {
    buffer: VecDeque<Transition>,
    capacity: usize,
}

impl EpisodicBuffer {
    pub fn new(capacity: usize) -> Self {
        Self {
            buffer: VecDeque::with_capacity(capacity),
            capacity,
        }
    }

    pub fn push(&mut self, transition: Transition) {
        if self.buffer.len() >= self.capacity {
            self.buffer.pop_front();
        }
        self.buffer.push_back(transition);
    }

    pub fn sample_prioritized(
        &self,
        priorities: &[f32],
        batch_size: usize,
        alpha: f32,
    ) -> (Vec<Transition>, Vec<f32>) {
        // Prioritized sampling implementation
```

```
        todo!()
    }
}
```

6.2.2 Schema Representation

```rust
use ndarray::{Array1, Array2};

#[derive(Clone, Debug, serde::Serialize, serde::Deserialize)]
pub struct Schema {
    pub id: uuid::Uuid,
    pub mean: Array1<f32>,
    pub covariance: Array2<f32>,
    pub num_points: usize,
    pub avg_reward: f32,
    pub created_at: chrono::DateTime<chrono::Utc>,
    pub last_updated: chrono::DateTime<chrono::Utc>,
}

impl Schema {
    pub fn likelihood(&self, state: &Array1<f32>) -> f32 {
        // Compute Gaussian likelihood
        let diff = state - &self.mean;
        let inv_cov = self.covariance.inv().unwrap();
        let exponent = -0.5 * diff.dot(&inv_cov.dot(&diff));
        exponent.exp()
    }

    pub fn update(
        &mut self,
        new_state: &Array1<f32>,
        learning_rate: f32,
    ) {
        let diff = new_state - &self.mean;
        self.mean = &self.mean + learning_rate * &diff;
        // Update covariance (outer product)
        let outer = diff.clone().insert_axis(Axis(1))
            .dot(&diff.clone().insert_axis(Axis(0)));
        self.covariance = (1.0 - learning_rate) * &self.covariance
            + learning_rate * outer;
        self.num_points += 1;
```

```
        self.last_updated = chrono::Utc::now();
    }
}
```

6.3 Error Handling

```
#[derive(Debug, thiserror::Error)]
pub enum BackwardError {
    #[error("Insufficient data for replay: {0} transitions")]
    InsufficientData(usize),

    #[error("Schema update failed: {0}")]
    SchemaUpdateError(String),

    #[error("UMAP fitting failed: {0}")]
    UmapError(String),

    #[error("Qdrant operation failed: {0}")]
    VectorDbError(#[from] qdrant_client::QdrantError),

    #[error("Linear algebra error: {0}")]
    LinalgError(String),
}

pub type BackwardResult<T> = Result<T, BackwardError>;
```

References

[1] Jordan Smith, "Project JANUS: Implementation Guide v1.0," 2025.

[2] Schaul et al., "Prioritized Experience Replay," ICLR 2016.

[3] "A Unified Dynamic Model for Learning, Replay, and Ripples," 2015/2025.

[4] Foster, Wilson, "Reverse Replay of Behavioural Sequences in Hippocampal Place Cells," Nature 2006.

[5] Tse et al., "Schema-Dependent Gene Activation and Memory Encoding in Neocortex," Science 2011.

[6] McInnes et al., "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," arXiv:1802.03426, 2018.

[7] Aynaud et al., "AlignedUMAP: Temporal Alignment for Multi-Dataset Visualization," bioRxiv, 2020.

[8] Qdrant Team, "Qdrant Vector Database Documentation," 2024.

JANUS
Neuromorphic Architecture

Brain-Inspired Algorithmic Trading System
Mapping Neuroscience to Market Intelligence

Classification: Architecture Specification

Version: 1.0

Author: Jordan Smith
github.com/nuniesmith

Date: December 26, 2025

Neuromorphic Design Principles:

- **Cognitive Mapping:** Each brain region maps to a trading subsystem
- **Hierarchical Processing:** From sensory input to strategic planning
- **Parallel Computation:** Multiple regions process simultaneously
- **Homeostatic Regulation:** Self-balancing risk and reward
- **Fear-Conditioned Safety:** Emotional override for threat response

Abstract

JANUS implements a **neuromorphic architecture** that maps cognitive neuroscience principles to algorithmic trading. Each brain region's computational role is replicated in the system architecture, creating a biologically-inspired trading intelligence that combines:

- **Cortex**: Strategic planning and long-term memory (Manager Agent)
- **Hippocampus**: Episodic memory and experience replay (Worker Agent)
- **Basal Ganglia**: Action selection via Actor-Critic RL
- **Thalamus**: Attention gating and multimodal fusion
- **Prefrontal Cortex**: Logic, planning, and regulatory compliance
- **Amygdala**: Fear detection and emergency circuit breakers
- **Hypothalamus**: Homeostatic risk regulation
- **Cerebellum**: Motor control and optimal execution
- **Visual Cortex**: Pattern recognition via GAF and ViViT
- **Integration**: Brainstem coordination and lifecycle management

This architecture enables emergent intelligence through parallel processing, hierarchical control, and homeostatic self-regulation—principles proven effective in biological systems over millions of years of evolution.

Contents

1 Neuromorphic Design Philosophy

1.1 Why Brain-Inspired Architecture?

Traditional trading systems follow rigid, hierarchical designs. JANUS instead adopts principles from cognitive neuroscience:

1. **Parallel Processing**: Multiple brain regions process different aspects of market data simultaneously
2. **Hierarchical Abstraction**: Low-level pattern recognition feeds mid-level tactics which inform high-level strategy
3. **Homeostatic Regulation**: Self-balancing mechanisms maintain system health (like biological homeostasis)
4. **Emotional Override**: Fear systems can immediately halt trading when threats are detected
5. **Memory Consolidation**: Wake-sleep cycles transfer episodic experiences to long-term schemas
6. **Adaptive Learning**: Continuous learning at multiple timescales (fast hippocampal, slow cortical)

1.2 Neuroscience-to-Trading Mapping

Brain Region	Neuroscience Function	Trading Function
Cortex	Executive function, strategic planning, declarative memory	Manager agent, portfolio strategy, market knowledge
Hippocampus	Episodic memory, spatial navigation, memory replay	Worker agent, trade history, experience replay
Basal Ganglia	Action selection, habit formation, reward learning	Actor-Critic RL, action gating, Q-learning
Thalamus	Sensory relay, attention gating, arousal	Data fusion, attention mechanisms, signal filtering
Prefrontal	Logic, planning, impulse control, ethics	LTN constraints, compliance, goal decomposition
Amygdala	Fear conditioning, threat detection, emotional memory	Risk detection, circuit breakers, kill switch
Hypothalamus	Homeostasis, motivation, energy balance	Risk appetite, position sizing, cash management
Cerebellum	Motor coordination, procedural learning, error correction	Order execution, slippage prediction, PID control
Visual Cortex	Visual processing, feature extraction, object recognition	GAF encoding, ViViT, pattern recognition
Brainstem	Basic life functions, arousal/sleep cycles	System orchestration, wake/sleep coordination

Table 1: Neuroscience to Trading Mapping

2 Brain Region Architectures

2.1 Cortex: Strategic Planning & Long-term Memory

2.1.1 Neuroscience Background

The cerebral cortex handles executive function, strategic planning, and declarative (fact-based) memory. It operates on slow timescales, consolidating knowledge over days to years.

2.1.2 Trading Implementation

Directory: `src/janus/neuromorphic/cortex/`

Components:

- **Manager**: Feudal RL manager agent for high-level strategy
- **Memory**: Long-term knowledge consolidation and schema storage
- **Planning**: Scenario analysis, Monte Carlo, portfolio optimization

Key Responsibilities:

1. Set strategic goals (e.g., "maximize Sharpe ratio while maintaining drawdown <15%")
2. Generate subgoals for Worker agent (e.g., "accumulate position in AAPL over 2 hours")
3. Consolidate episodic memories into abstract schemas (e.g., "morning volatility regime")
4. Store declarative knowledge (e.g., "FOMC announcements increase volatility")

Mathematical Formulation:
Manager policy selects subgoals g for Worker:

$$g_t = \pi_{\text{Manager}}(s_t^{\text{high}}) \tag{1}$$

where s_t^{high} is high-level state (portfolio metrics, regime, time-to-horizon).

Value function:

$$V_{\text{Manager}}(s) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t^{\text{high}} \mid s_0 = s\right] \tag{2}$$

2.2 Hippocampus: Episodic Memory & Experience Replay

2.2.1 Neuroscience Background

The hippocampus rapidly encodes episodic memories and replays them during sleep at 10-20$\times$ speed. Pattern separation prevents interference. Sharp Wave Ripples (SWR) prioritize important experiences.

2.2.2 Trading Implementation

Directory: `src/janus/neuromorphic/hippocampus/`

Components:

- **Worker**: Feudal RL worker agent for tactical execution
- **Replay**: Prioritized Experience Replay (PER) buffer

- **Episodes**: Trade sequences and market events
- **SWR**: Sharp Wave Ripple simulator for compressed replay

Key Responsibilities:

1. Execute subgoals from Manager (e.g., "buy 100 shares incrementally")
2. Store trade experiences in episodic buffer
3. Prioritize replay based on TD-error + logic violations
4. Compress replay 10-20× during sleep (Backward service)

Mathematical Formulation:
Worker policy conditioned on subgoal g:

$$a_t = \pi_{\text{Worker}}(s_t^{\text{low}}, g_t) \tag{3}$$

Intrinsic reward for subgoal completion:

$$r_t^{\text{intrinsic}} = -||s_t - g_t||^2 \tag{4}$$

Prioritized replay sampling:

$$P(i) = \frac{p_i^\alpha}{\sum_j p_j^\alpha}, \quad p_i = |\delta_i| + \lambda_{\text{logic}} v_i + \lambda_{\text{reward}} |r_i| \tag{5}$$

2.3 Basal Ganglia: Action Selection & Reinforcement Learning

2.3.1 Neuroscience Background

The basal ganglia implement action selection via dual pathways: direct (Go) promotes actions, indirect (No-Go) inhibits them. This is the biological substrate for reinforcement learning.

2.3.2 Trading Implementation

Directory: `src/janus/neuromorphic/basal_ganglia/`

Components:

- **Actor**: Policy network for action distribution
- **Critic**: Value network for advantage estimation
- **Praxeological**: Go/No-Go signal computation

- **Selection**: Competitive action selection mechanisms

Key Responsibilities:

1. Generate action proposals (BUY, SELL, HOLD, sizes)
2. Compute action values (Q-values)
3. Gate actions through dual pathways (safety)
4. Maintain habit cache for frequent patterns

Mathematical Formulation:

Actor policy:

$$\pi_\theta(a|s) = \text{softmax}(\mathbf{W}_\pi \mathbf{h}(s) + \mathbf{b}_\pi) \tag{6}$$

Critic value estimate:

$$V_\omega(s) = \mathbf{W}_V \mathbf{h}(s) + b_V \tag{7}$$

Advantage:

$$A(s,a) = Q(s,a) - V(s) \tag{8}$$

Go signal (direct pathway):

$$\text{Go}(a) = \max\left(\mathbf{W}_{\text{direct}} \mathbf{h}(s)\right)_a \tag{9}$$

No-Go signal (indirect pathway):

$$\text{NoGo}(a) = \text{sigmoid}(\mathbf{W}_{\text{indirect}}[\mathbf{h}(s); \text{risk}; \text{VPIN}]) \tag{10}$$

Final action gate:

$$a_{\text{final}} = \begin{cases} a_{\text{proposed}} & \text{if NoGo}(a) < \tau_{\text{veto}} \\ \text{HOLD} & \text{otherwise} \end{cases} \tag{11}$$

2.4 Thalamus: Attention & Multimodal Fusion

2.4.1 Neuroscience Background

The thalamus acts as a sensory relay station, gating information flow to cortex based on attention and relevance. It integrates multimodal sensory inputs.

2.4.2 Trading Implementation

Directory: `src/janus/neuromorphic/thalamus/`

Components:

- **Attention**: Cross-attention mechanisms
- **Gating**: Sensory gating and relevance filtering
- **Routing**: Dynamic information routing
- **Fusion**: Price, volume, orderbook, sentiment fusion

Key Responsibilities:

1. Gate incoming market data (filter noise)
2. Fuse multiple data modalities (price, volume, text)
3. Route relevant information to appropriate regions
4. Implement attention mechanisms for saliency

Mathematical Formulation:
Gated cross-attention:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \tag{12}$$

Gating scalar:

$$\lambda_{\text{gate}} = \text{sigmoid}(\mathbf{W}_g[\mathbf{h}_m; \mathbf{h}_n] + b_g) \tag{13}$$

Fused representation:

$$\mathbf{h}_{\text{fused}} = \mathbf{h}_m + \lambda_{\text{gate}} \cdot \text{Attention}(\mathbf{h}_m, \mathbf{h}_n, \mathbf{h}_n) \tag{14}$$

2.5 Prefrontal Cortex: Logic, Planning & Compliance

2.5.1 Neuroscience Background

The prefrontal cortex implements logical reasoning, impulse control, and ethical decision-making. It's the "executive" that enforces rules and long-term goals.

2.5.2 Trading Implementation

Directory: `src/janus/neuromorphic/prefrontal/`

Components:

- **LTN**: Logic Tensor Networks for rule encoding
- **Conscience**: Compliance constraints (wash sale, risk limits)
- **Planning**: Goal decomposition and plan synthesis

- **Goals**: Goal hierarchy management

Key Responsibilities:

1. Encode trading rules as differentiable logic
2. Enforce regulatory compliance (wash sale, position limits)
3. Block actions violating constraints
4. Decompose high-level goals into actionable plans

Mathematical Formulation:
LTN predicate grounding:

$$\mathcal{G}(P)(\mathbf{x}) = \text{sigmoid}(\mathbf{W}_P \mathbf{x} + b_P) \in [0, 1] \tag{15}$$

Łukasiewicz conjunction:

$$\mathcal{G}(A \wedge B) = \max(0, \mathcal{G}(A) + \mathcal{G}(B) - 1) \tag{16}$$

Wash sale constraint:

$$\forall t, k \in [1, 30] : \neg(\text{SaleAtLoss}(t) \wedge \text{Buy}(t + k)) \tag{17}$$

Satisfiability:

$$\text{SatAgg}(\mathcal{K}) = \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{G}(\phi_i)^p \right)^{1/p} \tag{18}$$

2.6 Amygdala: Fear, Threat Detection & Circuit Breakers

2.6.1 Neuroscience Background

The amygdala detects threats and triggers immediate fear responses, overriding rational planning when danger is present. Fear-conditioned memories persist long-term.

2.6.2 Trading Implementation

Directory: `src/janus/neuromorphic/amygdala/`

Components:

- **Fear**: Fear-conditioned inhibition network (FNI-RL)
- **VPIN**: Volume-synchronized toxicity detection
- **Circuit Breakers**: Kill switch, position freeze, cancel all

- **Threat Detection**: Anomaly, flash crash, black swan detection

Key Responsibilities:

1. Detect market panic and flash crashes
2. Trigger emergency circuit breakers
3. Override all other systems in extreme conditions
4. Learn fear-conditioned responses to past disasters

Mathematical Formulation:
VPIN (Volume-Synchronized Probability of Informed Trading):

$$\text{VPIN}_t = \frac{\sum_{i=1}^{n} |V_{\text{buy},i} - V_{\text{sell},i}|}{\sum_{i=1}^{n} V_i} \tag{19}$$

Fear activation:

$$f_{\text{fear}}(s) = \text{sigmoid}(\mathbf{W}_f[\text{VPIN}; \sigma_{\text{vol}}; \Delta p_{\text{max}}] + b_f) \tag{20}$$

Circuit breaker trigger:

$$\text{KillSwitch} = \begin{cases} \text{ACTIVATE} & \text{if } f_{\text{fear}} > \tau_{\text{fear}} \text{ OR VPIN} > \tau_{\text{VPIN}} \\ \text{STANDBY} & \text{otherwise} \end{cases} \tag{21}$$

2.7 Hypothalamus: Homeostasis & Risk Appetite

2.7.1 Neuroscience Background

The hypothalamus maintains homeostasis—internal balance of temperature, hunger, thirst, etc. It regulates motivation and energy expenditure.

2.7.2 Trading Implementation

Directory: `src/janus/neuromorphic/hypothalamus/`

Components:

- **Homeostasis**: Balance tracking and deviation correction
- **Position Sizing**: Kelly criterion, volatility scaling
- **Risk Appetite**: Dynamic risk tolerance adaptation
- **Energy**: Capital allocation and cash reserves

Key Responsibilities:

1. Maintain target portfolio balance (setpoints)
2. Adjust position sizes based on volatility and drawdown
3. Regulate risk appetite (fear vs. greed)
4. Ensure cash reserves and leverage limits

Mathematical Formulation:
Kelly criterion (fractional):

$$f^* = \frac{p(b+1)-1}{b}, \quad \text{position size} = \frac{f^*}{2} \cdot \text{capital} \tag{22}$$

Volatility scaling:

$$\text{size}_{\text{adjusted}} = \text{size}_{\text{base}} \cdot \frac{\sigma_{\text{target}}}{\sigma_{\text{current}}} \tag{23}$$

Drawdown scaling:

$$\text{size}_{\text{DD}} = \text{size}_{\text{base}} \cdot \max\left(0.1, 1 - \frac{\text{DD}_{\text{current}}}{\text{DD}_{\text{max}}}\right) \tag{24}$$

Homeostatic correction:

$$\Delta\text{allocation} = K_p \cdot (\text{target} - \text{current}) + K_d \cdot \frac{d(\text{target} - \text{current})}{dt} \tag{25}$$

2.8 Cerebellum: Motor Control & Execution

2.8.1 Neuroscience Background

The cerebellum coordinates fine motor control, learns procedural skills, and predicts sensory consequences of actions (forward models).

2.8.2 Trading Implementation

Directory: `src/janus/neuromorphic/cerebellum/`

Components:

- **Execution**: Order routing, TWAP/VWAP algorithms
- **Impact**: Almgren-Chriss optimal execution
- **Forward Models**: Latency compensation, fill prediction
- **Error Correction**: PID control, adaptive feedback

Key Responsibilities:

1. Route orders to exchanges with minimal slippage
2. Predict and minimize market impact
3. Compensate for execution latency (Smith predictor)
4. Learn from execution errors and adapt

Mathematical Formulation:
Almgren-Chriss optimal trajectory:

$$x_t = X \cdot \frac{\sinh(\kappa(T-t))}{\sinh(\kappa T)}, \quad \kappa = \sqrt{\frac{\eta\sigma}{\tau}} \tag{26}$$

Market impact:

$$\text{Impact} = \eta \cdot \sigma \cdot \sqrt{\frac{v}{V_{\text{avg}}}} \tag{27}$$

Smith predictor (latency compensation):

$$u(t) = K_c \left[e(t) + \frac{1}{\tau_I} \int e(\tau) d\tau + \tau_D \frac{de(t)}{dt} \right] + \hat{p}(t + \Delta t) \tag{28}$$

Execution error:

$$\epsilon_{\text{exec}} = |p_{\text{actual}} - p_{\text{predicted}}| \tag{29}$$

2.9 Visual Cortex: Pattern Recognition & Vision

2.9.1 Neuroscience Background

The visual cortex processes images hierarchically: V1 detects edges, V2 detects shapes, V4 detects objects. It implements hierarchical feature extraction.

2.9.2 Trading Implementation

Directory: `src/janus/neuromorphic/visual_cortex/`

Components:

- **Eyes**: Data ingestion, preprocessing, streaming
- **GAF**: Gramian Angular Fields (GASF, GADF, DiffGAF)
- **ViViT**: Video Vision Transformer for spatiotemporal patterns
- **Visualization**: UMAP, GradCAM, saliency maps

Key Responsibilities:

1. Ingest and preprocess raw market data
2. Transform time series to visual manifolds (GAF)
3. Extract spatiotemporal patterns (ViViT)
4. Visualize learned representations (UMAP)

Mathematical Formulation:
GAF normalization:

$$\tilde{x}_i = \tanh\left(\frac{x_i - \min(X)}{\max(X) - \min(X) + \epsilon} \cdot \alpha + \beta\right) \tag{30}$$

GASF:

$$\text{GASF}_{i,j} = \cos(\phi_i + \phi_j), \quad \phi_i = \arccos(\tilde{x}_i) \tag{31}$$

GADF:

$$\text{GADF}_{i,j} = \sin(\phi_i - \phi_j) \tag{32}$$

ViViT patch embedding:

$$\mathbf{z}_{f,i,j}^{(0)} = \mathbf{E} \cdot \text{flatten}(\mathcal{V}_{f,i:i+P,j:j+P}) + \mathbf{p}_{f,i,j} \tag{33}$$

2.10 Integration: Brainstem & Global Coordination

2.10.1 Neuroscience Background

The brainstem controls basic life functions, arousal/sleep cycles, and global state coordination. It's the "operating system" of the brain.

2.10.2 Trading Implementation

Directory: `src/janus/neuromorphic/integration/`

Components:

- **Workflow**: State machine orchestration
- **State**: Global state management, message bus
- **API**: REST, gRPC, WebSocket interfaces
- **Engine**: Wake-sleep cycle coordination

Key Responsibilities:

1. Coordinate wake (Forward) and sleep (Backward) cycles

2. Manage global system state
3. Route messages between brain regions
4. Expose external APIs

3 Information Flow Diagrams

3.1 Wake State (Forward Service)

```
Market Data → Visual Cortex (GAF/ViViT) → Thalamus (Fusion)
                                                ↓
                                          Cortex (Manager)
                                                ↓
                                          Hippocampus (Worker)
                                                ↓
                                          Basal Ganglia (Actor-Critic)
                                                ↓
                                          Prefrontal (LTN Check)
                                                ↓
                                          Amygdala (Fear Check) —→ Circuit Breaker?
                                                ↓
                                          Hypothalamus (Size Adjust)
                                                ↓
                                          Cerebellum (Execute Order)
                                                ↓
                                          Exchange
```

3.2 Sleep State (Backward Service)

```
Hippocampus (Episodic Buffer) → SWR Replay (10-20x speed)
                                      ↓
                                Prioritized Sampling
                                      ↓
                                Basal Ganglia (Update Critic)
                                      ↓
                                Prefrontal (Update LTN)
                                      ↓
                                Cortex (Schema Consolidation)
                                      ↓
                                Long-term Memory (Qdrant)
```

4 Implementation Guide

4.1 Directory Structure

```
src/janus/neuromorphic/
├── lib.rs                    # Main library entry point
├── common/                   # Shared types and utilities
├── cortex/                   # Strategic planning & LTM
│   ├── manager/              # Feudal RL manager
│   ├── memory/               # Consolidation, schemas
│   └── planning/             # Scenario analysis
├── hippocampus/              # Episodic memory & replay
│   ├── worker/               # Feudal RL worker
│   ├── replay/               # PER buffer
│   ├── episodes/             # Trade sequences
│   └── swr/                  # Sharp wave ripples
├── basal_ganglia/            # Action selection & RL
│   ├── actor/                # Policy network
│   ├── critic/               # Value network
│   ├── praxeological/        # Go/No-Go signals
│   └── selection/            # Action selection
├── thalamus/                 # Attention & fusion
│   ├── attention/            # Cross-attention
│   ├── gating/               # Sensory gates
│   ├── routing/              # Information routing
│   └── fusion/               # Multimodal fusion
├── prefrontal/               # Logic & compliance
│   ├── ltn/                  # Logic Tensor Networks
│   ├── conscience/           # Compliance rules
│   ├── planning/             # Goal decomposition
│   └── goals/                # Goal management
├── amygdala/                 # Fear & circuit breakers
│   ├── fear/                 # FNI-RL network
│   ├── vpin/                 # Toxicity detection
│   ├── circuit_breakers/     # Kill switch
│   └── threat_detection/     # Anomaly detection
├── hypothalamus/             # Homeostasis & risk
│   ├── homeostasis/          # Balance tracking
│   ├── position_sizing/      # Kelly, vol scaling
```

```
    ├── risk_appetite/       # Dynamic tolerance
    └── energy/              # Capital allocation
├── cerebellum/              # Motor control & execution
    ├── execution/           # Order routing
    ├── impact/              # Almgren-Chriss
    ├── forward_models/      # Latency compensation
    └── error_correction/    # PID control
├── visual_cortex/           # Pattern recognition
    ├── eyes/                # Data ingestion
    ├── gaf/                 # GAF transformation
    ├── vivit/               # ViViT model
    └── visualization/       # UMAP, GradCAM
└── integration/             # System coordination
    ├── workflow/            # State machines
    ├── state/               # Global state
    ├── api/                 # External APIs
    └── engine/              # Orchestration
```

4.2 Implementation Checklist

1. **Phase 1: Core Infrastructure (Weeks 1-2)**
   - ☐ Set up neuromorphic module structure
   - ☐ Implement common types and error handling
   - ☐ Create inter-region message bus
   - ☐ Set up integration/engine orchestrator
2. **Phase 2: Visual Processing (Weeks 3-4)**
   - ☐ Implement Visual Cortex data ingestion
   - ☐ Implement GAF transformation (GASF, GADF)
   - ☐ Integrate ViViT model (ONNX or tch-rs)
   - ☐ Add UMAP visualization
3. **Phase 3: Decision Making (Weeks 5-7)**
   - ☐ Implement Basal Ganglia Actor-Critic
   - ☐ Implement Prefrontal LTN constraints
   - ☐ Implement Thalamus fusion mechanisms

- ☐ Connect visual → decision pipeline

4. **Phase 4: Memory Systems (Weeks 8-10)**
   - ☐ Implement Hippocampus episodic buffer
   - ☐ Implement Prioritized Experience Replay
   - ☐ Implement Cortex schema consolidation
   - ☐ Implement SWR compressed replay
5. **Phase 5: Safety & Control (Weeks 11-12)**
   - ☐ Implement Amygdala fear network
   - ☐ Implement circuit breakers and kill switch
   - ☐ Implement Hypothalamus homeostasis
   - ☐ Implement Cerebellum execution control
6. **Phase 6: Integration & Testing (Weeks 13-14)**
   - ☐ Connect all brain regions
   - ☐ Implement wake-sleep cycle coordination
   - ☐ End-to-end integration tests
   - ☐ Performance optimization

5 Architectural Invariants

5.1 Safety-Critical Invariants

1. **Amygdala Override**: Fear system ALWAYS overrides all other regions
2. **Prefrontal Veto**: LTN constraints MUST block non-compliant actions
3. **No Panic**: No `panic!()`, `unwrap()`, or `expect()` in production code
4. **Fail-Safe**: Circuit breakers must be fail-safe (default to HALT)
5. **Homeostasis**: Cash reserves must never fall below 20%

5.2 Performance Invariants

1. **Forward Latency**: Visual Cortex → Decision <100ms
2. **Backward Throughput**: Process >10k experiences per sleep cycle
3. **Memory Efficiency**: Hippocampal buffer <10k transitions (FIFO eviction)
4. **Parallel Processing**: Brain regions process concurrently

5.3 Learning Invariants

1. **Dual Timescale**: Fast hippocampal learning, slow cortical consolidation
2. **Recall Gating**: Cortical updates gated by recall strength AND logic validity
3. **Priority Replay**: Replay prioritized by TD-error + logic violations + reward
4. **Schema Formation**: Clusters detected via UMAP + DBSCAN

References

[1] Jordan Smith, "JANUS Forward: Wake State Logic Trading Algorithm," 2025.

[2] Jordan Smith, "JANUS Backward: Sleep State Memory Management," 2025.

[3] Dayan, Hinton, "Feudal Reinforcement Learning," NIPS 1992.

[4] Sutton, Barto, "Reinforcement Learning: An Introduction," 2nd Ed., 2018.

[5] Badreddine et al., "Logic Tensor Networks," Artificial Intelligence, 2022.

[6] Schaul et al., "Prioritized Experience Replay," ICLR 2016.

[7] Foster, Wilson, "Reverse Replay of Behavioural Sequences in Hippocampal Place Cells," Nature 2006.

[8] "Fear-Conditioned Inhibition in Reinforcement Learning," 2020.

[9] Sterling, Eyer, "Allostasis: A New Paradigm to Explain Arousal Pathology," 1988.

[10] Almgren, Chriss, "Optimal Execution of Portfolio Transactions," Journal of Risk, 2000.

[11] Easley et al., "Flow Toxicity and Liquidity in a High-frequency World," Review of Financial Studies, 2012.

[12] Wang, Oates, "Encoding Time Series as Images," AAAI 2015.

[13] Arnab et al., "ViViT: A Video Vision Transformer," ICCV 2021.

[14] McInnes et al., "UMAP: Uniform Manifold Approximation and Projection," 2018.

[15] Kandel et al., "Principles of Neural Science," 6th Ed., 2021.

JANUS
Rust-First ML Implementation

End-to-End Machine Learning with Rust
FastAPI Gateway & Batch Processing Architecture

Classification: Architecture & Implementation Strategy

Version: 1.0

Author: Jordan Smith
github.com/nuniesmith

Date: December 26, 2025

Migration Philosophy:

- **Core ML in Rust:** Maximum performance, safety, and type guarantees
- **Python Gateway:** FastAPI for orchestration and API surface
- **Hybrid Approach:** Use best tool for each layer
- **Modern Stack:** Latest technologies (Candle, ONNX, Burn, tch-rs)

Abstract

This document outlines a comprehensive strategy for implementing the JANUS trading system with a **Rust-first approach to machine learning**, while maintaining a Python FastAPI gateway for orchestration and external API exposure. The architecture leverages Rust's performance, memory safety, and type system for the entire ML pipeline—from data preprocessing to inference—while using Python strategically for high-level coordination, batch job scheduling, and developer-friendly interfaces.

Key architectural decisions:

- **Hot path (Forward):** Pure Rust with ONNX Runtime or tch-rs for <100ms latency
- **Cold path (Backward):** Rust core with optional Python for experimentation
- **Gateway:** FastAPI service exposing REST/gRPC APIs
- **Batch processing:** Rust workers with async job queue
- **Training:** Hybrid Python (PyTorch) to Rust (ONNX/tch-rs) pipeline

This approach maximizes performance and safety while maintaining ecosystem compatibility and developer productivity.

Contents

1 Architectural Overview

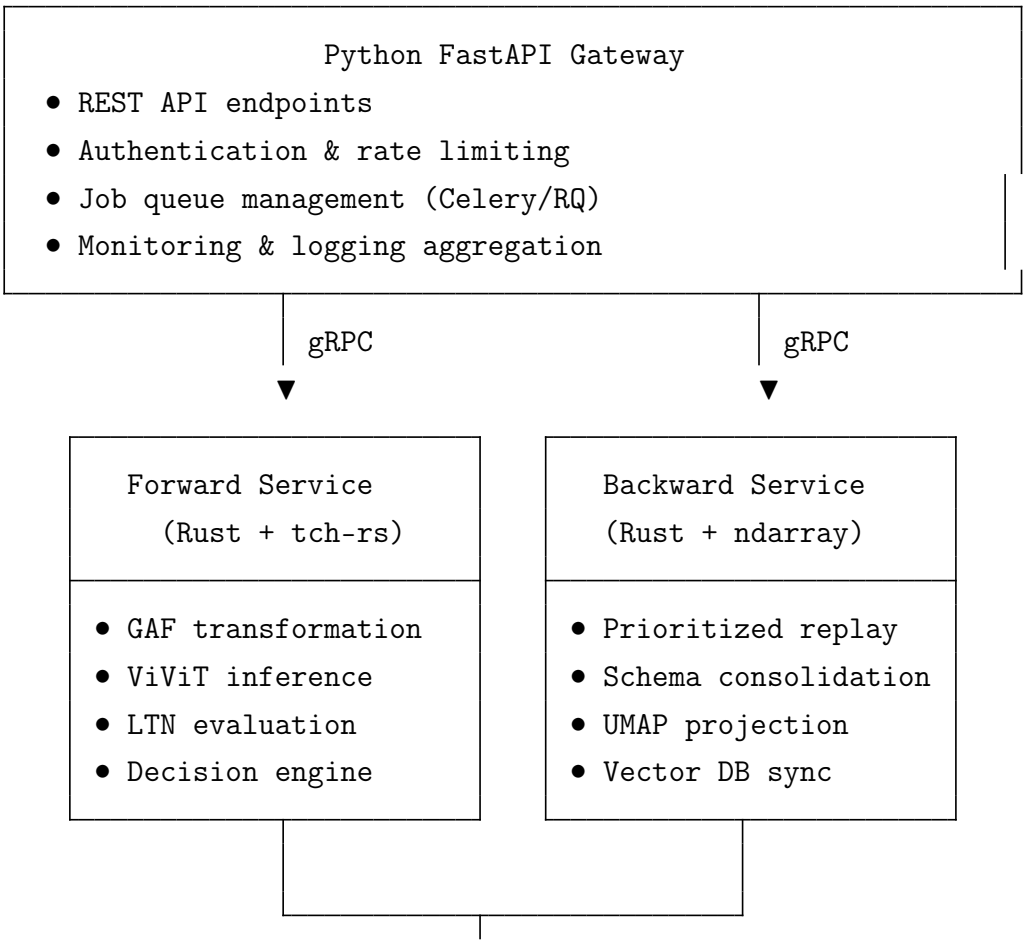
1.1 The Rust-First Philosophy

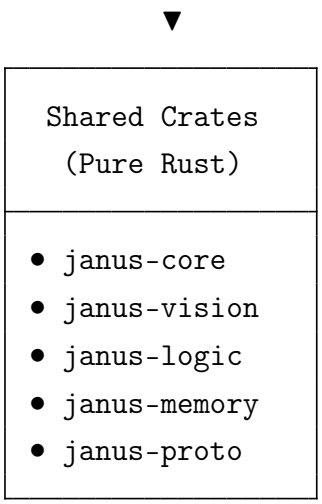
Traditional ML systems place Python at the center, with C++/Rust used only for performance-critical bottlenecks. JANUS inverts this model:

JANUS Layering

1. **Core Layer (Rust):** All ML inference, data processing, logic evaluation
2. **Service Layer (Rust):** gRPC services for Forward and Backward
3. **Gateway Layer (Python FastAPI):** HTTP API, authentication, rate limiting
4. **Training Layer (Hybrid):** PyTorch training → ONNX/tch export → Rust inference

1.2 Component Diagram





1.3 Design Principles

1. **Rust Everywhere Possible**
   - All inference, data transformation, and business logic in Rust
   - Zero-copy operations where possible
   - Compile-time guarantees for correctness
2. **Python as Orchestration Layer**
   - FastAPI for HTTP endpoints and developer UX
   - Async job scheduling (Celery, Dramatiq, or Python RQ)
   - High-level monitoring and alerting
3. **gRPC for Internal Communication**
   - Type-safe, efficient service-to-service communication
   - Shared protobuf definitions in `janus-proto`
   - Streaming support for real-time data
4. **Gradual Migration Strategy**
   - Start with inference in Rust (ONNX models)
   - Migrate data preprocessing incrementally
   - Keep training in PyTorch initially, export to Rust
   - Eventually move training to Rust (Burn/Candle)

2 Machine Learning Framework Strategy

2.1 Framework Comparison Matrix

Framework	Pros	Cons	Use Case
tch-rs	Full PyTorch bindings, mature, GPU support	Requires LibTorch, C++ dependency	**Phase 1: Inference**
ONNX Runtime	Lightweight, optimized, cross-platform	Inference only, limited operators	**Phase 1: Production**
Candle (HF)	Pure Rust, no C++ deps, growing ecosystem	Young, fewer pre-trained models	**Phase 2: Long-term**
Burn	Backend-agnostic, autodiff, training support	Early stage, limited models	**Phase 3: Full Rust**
ndarray	NumPy-like API, stable, CPU optimized	No GPU, no autodiff	**Data processing**

Table 1: Rust ML Framework Comparison

2.2 Recommended Migration Path

2.2.1 Phase 1: Hybrid (Months 1-3)

- **Training:** PyTorch on GPU cluster
- **Export:** `torch.onnx.export()` or `torch.jit.save()`
- **Inference:** ONNX Runtime (`ort`) or tch-rs in Rust services
- **Why:** Fastest path to production, leverage existing PyTorch ecosystem

2.2.2 Phase 2: Rust-Native Inference (Months 4-6)

- **Training:** Still PyTorch
- **Export:** Candle-compatible checkpoints
- **Inference:** Candle or custom Rust implementations
- **Why:** Eliminate LibTorch dependency, reduce binary size

2.2.3 Phase 3: Full Rust ML (Months 7-12)

- **Training:** Burn or Candle with custom training loops
- **Inference:** Same framework as training
- **Why:** Complete type safety, no Python GIL, full control

2.3 Framework Selection by Component

Component	Framework (Phase 1)	Framework (Phase 3)
GAF Transformation	`ndarray`	`ndarray` or `candle`
ViViT Inference	`ort` (ONNX)	`candle`
LTN Predicates	`tch-rs`	`candle`
Chronos-Bolt	`ort` (ONNX)	`candle`
FinBERT	`ort` (ONNX)	`candle-transformers`
Decision Engine	`tch-rs`	`burn`
PER Buffer	Custom Rust	Custom Rust
UMAP	`linfa-reduction`	`linfa-reduction`

Table 2: Component-Level Framework Selection

3 Forward Service: Rust Implementation

3.1 Performance Requirements

- **End-to-end latency:** <100ms (target: 50ms)
- **Throughput:** >100 requests/second
- **Memory:** <2GB per instance
- **CPU:** Efficient use of multi-core (no GIL)

3.2 Core Data Structures

```rust
// janus-core/src/types.rs
use ndarray::{Array1, Array2, Array3};

#[derive(Clone, Debug, serde::Serialize, serde::Deserialize)]
pub struct MarketState {
    pub timestamp: i64,
```

```
    pub price_series: Array1<f32>,  // Last N prices
    pub volume_series: Array1<f32>, // Last N volumes
    pub lob_snapshot: LimitOrderBook,
    pub vpin: f32,
    pub volatility: f32,
}

#[derive(Clone, Debug)]
pub struct LimitOrderBook {
    pub bids: Vec<(f32, f32)>,  // (price, size)
    pub asks: Vec<(f32, f32)>,
}

#[derive(Clone, Debug)]
pub enum Action {
    Buy { size: f32 },
    Sell { size: f32 },
    Hold,
}

#[derive(Clone, Debug)]
pub struct Decision {
    pub action: Action,
    pub confidence: f32,
    pub ltn_satisfaction: f32,
    pub risk_score: f32,
    pub metadata: HashMap<String, f32>,
}
```

3.3 GAF Transformation Module

```
// janus-vision/src/gaf.rs
use ndarray::{Array1, Array2};

pub struct GafTransformer {
    alpha: f32,  // Learnable normalization param
    beta: f32,   // Learnable normalization param
}

impl GafTransformer {
```

```rust
    pub fn transform(&self, series: &Array1<f32>) -> (Array2<f32>, Array2<f32>) {
        // Step 1: Normalize to [-1, 1]
        let min = series.iter().cloned().fold(f32::INFINITY, f32::min);
        let max = series.iter().cloned().fold(f32::NEG_INFINITY, f32::max);
        let normalized = series.mapv(|x| {
            let norm = (x - min) / (max - min + 1e-8);
            (norm * self.alpha + self.beta).tanh()
        });

        // Step 2: Polar coordinates
        let phi = normalized.mapv(|x| x.acos());

        // Step 3: Gramian fields
        let n = series.len();
        let mut gasf = Array2::<f32>::zeros((n, n));
        let mut gadf = Array2::<f32>::zeros((n, n));

        for i in 0..n {
            for j in 0..n {
                gasf[[i, j]] = (phi[i] + phi[j]).cos();
                gadf[[i, j]] = (phi[i] - phi[j]).sin();
            }
        }

        (gasf, gadf)
    }

    pub fn video(&self, series: &Array1<f32>, window: usize, stride: usize, frames: usize)
        -> Array3<f32>
    {
        let mut video_frames = Vec::new();

        for f in 0..frames {
            let start = f * stride;
            let end = start + window;
            if end > series.len() {
                break;
```

```
            }
            let window_series = series.slice(s![start..end]).
                to_owned();
            let (gasf, gadf) = self.transform(&window_series);

            // Stack GASF and GADF as channels
            video_frames.push(gasf);
            video_frames.push(gadf);
        }

        // Stack into 3D tensor: [frames, 2, window, window]
        stack_frames(&video_frames, frames, window)
    }
}
```

3.4 LTN Constraint Evaluation

```
// janus-logic/src/ltn.rs
use tch::{Tensor, nn};

pub struct LogicTensorNetwork {
    predicates: HashMap<String, PredicateNet>,
}

pub struct PredicateNet {
    model: nn::Sequential,
}

impl PredicateNet {
    pub fn evaluate(&self, embedding: &Tensor) -> f32 {
        let logit = self.model.forward(embedding);
        logit.sigmoid().double_value(&[]) as f32
    }
}

impl LogicTensorNetwork {
    pub fn check_wash_sale(&self, action: &Action, history: &[
        Action]) -> f32 {
        // Encode wash sale constraint
        //   t  ,  k  in [1,30]: not(SaleAtLoss(t) and Buy(t+k))
        let mut min_satisfaction = 1.0;
```

```
24
25          // Implementation of  ukasiewicz  t-norm
26          for (t, past_action) in history.iter().enumerate() {
27              if let Action::Sell { .. } = past_action {
28                  if is_loss(past_action) {
29                      for k in 1..=30 {
30                          if t + k < history.len() {
31                              if let Action::Buy { .. } = &history[t
                                    + k] {
32                                  //  ukasiewicz  conjunction: max(0,
                                        a + b - 1)
33                                  let conjunction = (1.0 + 1.0 - 1.0)
                                        .max(0.0);
34                                  // Negation: 1 - conjunction
35                                  let satisfaction = 1.0 -
                                        conjunction;
36                                  min_satisfaction = min_satisfaction
                                        .min(satisfaction);
37                              }
38                          }
39                      }
40                  }
41              }
42          }
43
44          min_satisfaction
45      }
46
47      pub fn check_almgren_chriss(&self, state: &MarketState, action:
            &Action) -> f32 {
48          let volatility = state.volatility;
49          let avg_volume = state.volume_series.mean().unwrap();
50
51          match action {
52              Action::Buy { size } | Action::Sell { size } => {
53                  let eta = 0.1;  // Market impact coefficient
54                  let threshold = eta * volatility * (size /
                        avg_volume).sqrt();
55
56                  // If impact < threshold, satisfaction = 1.0
57                  // Smooth approximation with sigmoid
```

```
                let impact_ratio = size / (avg_volume * threshold);
                1.0 - impact_ratio.min(1.0)
            }
            Action::Hold => 1.0,
        }
    }

    pub fn check_vpin(&self, state: &MarketState) -> f32 {
        let vpin_threshold = 0.7;

        // If VPIN > threshold, should halt trading
        if state.vpin > vpin_threshold {
            0.0  // Constraint violated
        } else {
            1.0  // Constraint satisfied
        }
    }

    pub fn evaluate_all(&self, state: &MarketState, action: &Action
        , history: &[Action]) -> f32 {
        let wash_sale = self.check_wash_sale(action, history);
        let almgren = self.check_almgren_chriss(state, action);
        let vpin = self.check_vpin(state);

        // Generalized mean (p=2 for quadratic mean)
        let constraints = vec![wash_sale, almgren, vpin];
        let sum_squared: f32 = constraints.iter().map(|x| x.powi(2)
           ).sum();
        (sum_squared / constraints.len() as f32).sqrt()
    }
}
```

3.5 Model Inference with ONNX Runtime

```
// janus-vision/src/vivit.rs
use ort::{Environment, Session, SessionBuilder, Value};
use ndarray::Array4;

pub struct ViViTInference {
    session: Session,
    env: Arc<Environment>,
```

```rust
}

impl ViViTInference {
    pub fn new(model_path: &str) -> Result<Self, ort::OrtError> {
        let env = Arc::new(Environment::builder().build()?);
        let session = SessionBuilder::new(&env)?
            .with_optimization_level(ort::GraphOptimizationLevel::
               Level3)?
            .with_intra_threads(4)?
            .with_model_from_file(model_path)?;

        Ok(Self { session, env })
    }

    pub fn infer(&self, video: &Array4<f32>) -> Result<Array1<f32>,
        ort::OrtError> {
        // Convert ndarray to ONNX tensor
        let input_tensor = Value::from_array(self.session.allocator
           (), video)?;

        // Run inference
        let outputs = self.session.run(vec![input_tensor])?;

        // Extract output
        let output_tensor = outputs[0].try_extract()?;
        let embedding = output_tensor.view().to_owned();

        Ok(embedding)
    }
}
```

3.6 Async Service with Tokio

```rust
// services/forward/src/main.rs
use tonic::{transport::Server, Request, Response, Status};
use janus_proto::forward_service_server::{ForwardService,
   ForwardServiceServer};
use janus_proto::{DecisionRequest, DecisionResponse};

pub struct ForwardServiceImpl {
    gaf_transformer: Arc<GafTransformer>,
```

```
    vivit_model: Arc<ViViTInference>,
    ltn: Arc<LogicTensorNetwork>,
    decision_engine: Arc<DecisionEngine>,
}

#[tonic::async_trait]
impl ForwardService for ForwardServiceImpl {
    async fn get_decision(
        &self,
        request: Request<DecisionRequest>,
    ) -> Result<Response<DecisionResponse>, Status> {
        let req = request.into_inner();

        // Parse market state
        let state = parse_market_state(&req)?;

        // 1. GAF transformation
        let video = self.gaf_transformer.video(
            &state.price_series,
            60,  // window
            10,  // stride
            16,  // frames
        );

        // 2. ViViT inference
        let visual_embedding = self.vivit_model.infer(&video)
            .map_err(|e| Status::internal(format!("ViViT inference
               failed: {}", e)))?;

        // 3. LTN evaluation
        let ltn_satisfaction = self.ltn.evaluate_all(&state, &
           Action::Hold, &[]);

        // 4. Decision engine
        let decision = self.decision_engine.decide(
            &state,
            &visual_embedding,
            ltn_satisfaction,
        )?;

        // 5. Build response
```

```
        let response = DecisionResponse {
            action: decision.action.to_string(),
            confidence: decision.confidence,
            ltn_satisfaction: decision.ltn_satisfaction,
            risk_score: decision.risk_score,
            metadata: decision.metadata,
        };

        Ok(Response::new(response))
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let addr = "0.0.0.0:50051".parse()?;

    let service = ForwardServiceImpl {
        gaf_transformer: Arc::new(GafTransformer::new(1.0, 0.0)),
        vivit_model: Arc::new(ViViTInference::new("models/vivit.
            onnx")?),
        ltn: Arc::new(LogicTensorNetwork::load("models/ltn")?),
        decision_engine: Arc::new(DecisionEngine::new()),
    };

    println!("Forward service listening on {}", addr);

    Server::builder()
        .add_service(ForwardServiceServer::new(service))
        .serve(addr)
        .await?;

    Ok(())
}
```

4 Backward Service: Batch Processing

4.1 Performance Requirements

- **Latency:** Not critical (batch processing)
- **Throughput:** Process 10k-100k transitions per sleep cycle

- **Memory:** Can use up to 16GB for large batches
- **Parallelism:** Leverage all CPU cores with Rayon

4.2 Prioritized Experience Replay

```rust
// janus-memory/src/replay.rs
use rayon::prelude::*;
use rand::distributions::WeightedIndex;
use rand::prelude::*;

pub struct PrioritizedReplayBuffer {
    buffer: Vec<Transition>,
    priorities: Vec<f32>,
    capacity: usize,
    alpha: f32,  // Prioritization exponent
    beta: f32,   // Importance sampling correction
}

impl PrioritizedReplayBuffer {
    pub fn sample(&self, batch_size: usize) -> (Vec<Transition>, Vec<f32>) {
        let probabilities: Vec<f32> = self.priorities
            .iter()
            .map(|p| p.powf(self.alpha))
            .collect();

        let total: f32 = probabilities.iter().sum();
        let normalized_probs: Vec<f32> = probabilities
            .iter()
            .map(|p| p / total)
            .collect();

        let dist = WeightedIndex::new(&normalized_probs).unwrap();
        let mut rng = thread_rng();

        let indices: Vec<usize> = (0..batch_size)
            .map(|_| dist.sample(&mut rng))
            .collect();

        // Compute importance sampling weights
        let n = self.buffer.len() as f32;
```

```
        let weights: Vec<f32> = indices
            .iter()
            .map(|&i| {
                let prob = normalized_probs[i];
                ((1.0 / n) * (1.0 / prob)).powf(self.beta)
            })
            .collect();

        // Normalize weights
        let max_weight = weights.iter().cloned().fold(f32::
           NEG_INFINITY, f32::max);
        let normalized_weights: Vec<f32> = weights
            .iter()
            .map(|w| w / max_weight)
            .collect();

        let transitions: Vec<Transition> = indices
            .iter()
            .map(|&i| self.buffer[i].clone())
            .collect();

        (transitions, normalized_weights)
    }

    pub fn update_priorities(&mut self, indices: Vec<usize>,
       td_errors: Vec<f32>,
                             logic_violations: Vec<f32>, rewards:
                                Vec<f32>) {
        for (idx, (&i, (&td, (&vio, &rew)))) in indices.iter()
            .zip(td_errors.iter()
                .zip(logic_violations.iter()
                    .zip(rewards.iter())))
            .enumerate()
        {
            // Priority = |TD-error| +  _logic  * violation +
                _reward  * |reward|
            self.priorities[i] = td.abs() + 2.0 * vio + 0.5 * rew.
               abs();
        }
    }
}
```

4.3 Schema Consolidation

```rust
// janus-memory/src/schema.rs
use ndarray::{Array1, Array2};

pub struct SchemaMemory {
    schemas: Vec<Schema>,
    recall_threshold: f32,
    logic_threshold: f32,
}

impl SchemaMemory {
    pub fn consolidate(&mut self, transitions: &[Transition], ltn: &LogicTensorNetwork) {
        for transition in transitions {
            // Compute gates
            let recall_gate = self.compute_recall_gate(&transition.state);
            let logic_gate = ltn.evaluate_all(&transition.state, &transition.action, &[]);

            // Only consolidate if both gates pass
            if recall_gate > self.recall_threshold && logic_gate > self.logic_threshold {
                // Find best matching schema
                let (best_idx, best_likelihood) = self.find_best_schema(&transition.state);

                if best_likelihood < 0.01 {
                    // Create new schema
                    self.create_schema(&transition.state);
                } else {
                    // Update existing schema
                    self.update_schema(best_idx, &transition.state, 0.01);
                }
            }
        }
    }

    fn update_schema(&mut self, idx: usize, state: &Array1<f32>, lr: f32) {
```

```
        let schema = &mut self.schemas[idx];

        // Update mean:    = (1-  )   +   *x
        let diff = state - &schema.mean;
        schema.mean = &schema.mean + lr * &diff;

        // Update covariance:    = (1-  )   +   *(x-  )(x-  )
        let outer = outer_product(&diff, &diff);
        schema.covariance = (1.0 - lr) * &schema.covariance + lr *
           outer;

        schema.num_points += 1;
    }
}
```

4.4 Parallel Sleep Cycle

```
// services/backward/src/sleep_cycle.rs
use rayon::prelude::*;

pub async fn run_sleep_cycle(
    replay_buffer: &mut PrioritizedReplayBuffer,
    schema_memory: &mut SchemaMemory,
    ltn: &LogicTensorNetwork,
    policy: &mut Policy,
) -> Result<SleepCycleMetrics, BackwardError> {
    let start = std::time::Instant::now();
    let mut metrics = SleepCycleMetrics::default();

    // Phase 1: Prioritized Replay (1000 iterations)
    for iteration in 0..1000 {
        let (batch, weights) = replay_buffer.sample(256);

        // Parallel TD-error computation
        let td_errors: Vec<f32> = batch.par_iter()
            .map(|transition| compute_td_error(transition, policy))
            .collect();

        // Parallel logic violation scoring
        let violations: Vec<f32> = batch.par_iter()
            .map(|transition| {
```

```
                1.0 - ltn.evaluate_all(&transition.state, &
                    transition.action, &[])
            })
            .collect();

        // Update policy with importance-weighted gradients
        policy.update(&batch, &weights);

        // Update priorities
        let rewards: Vec<f32> = batch.iter().map(|t| t.reward).
            collect();
        let indices: Vec<usize> = (0..batch.len()).collect();
        replay_buffer.update_priorities(indices, td_errors,
            violations, rewards);

        if iteration % 100 == 0 {
            println!("Replay iteration {}/1000", iteration);
        }
    }

    // Phase 2: Schema Consolidation
    let all_transitions: Vec<Transition> = replay_buffer.buffer.
        clone();
    schema_memory.consolidate(&all_transitions, ltn);
    metrics.num_schemas = schema_memory.schemas.len();

    // Phase 3: UMAP Update
    let embeddings = extract_schema_embeddings(schema_memory);
    let umap_projection = fit_aligned_umap(&embeddings)?;
    metrics.num_clusters = detect_clusters(&umap_projection);

    // Phase 4: Vector DB Sync
    sync_to_qdrant(schema_memory).await?;

    metrics.duration = start.elapsed();
    Ok(metrics)
```

5 Python FastAPI Gateway

5.1 Architecture Purpose

The Python gateway serves as:

- **API Surface:** User-friendly REST endpoints
- **Authentication:** JWT tokens, API key validation
- **Rate Limiting:** Per-user request throttling
- **Job Scheduling:** Async batch processing with Celery/Dramatiq
- **Monitoring:** Metrics aggregation and alerting

5.2 FastAPI Service Implementation

```python
# gateway/main.py
from fastapi import FastAPI, Depends, HTTPException, BackgroundTasks
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import grpc
from pydantic import BaseModel
import asyncio

# Generated from proto files
from janus_proto import forward_service_pb2, forward_service_pb2_grpc
from janus_proto import backward_service_pb2, backward_service_pb2_grpc

app = FastAPI(title="JANUS Trading System API")
security = HTTPBearer()

# gRPC channel pool
forward_channel = grpc.aio.insecure_channel('forward-service:50051')
backward_channel = grpc.aio.insecure_channel('backward-service:50052')

forward_stub = forward_service_pb2_grpc.ForwardServiceStub(forward_channel)
backward_stub = backward_service_pb2_grpc.BackwardServiceStub(backward_channel)
```

```python

class MarketStateRequest(BaseModel):
    timestamp: int
    price_series: list[float]
    volume_series: list[float]
    lob_bids: list[tuple[float, float]]
    lob_asks: list[tuple[float, float]]

class DecisionResponse(BaseModel):
    action: str
    confidence: float
    ltn_satisfaction: float
    risk_score: float
    latency_ms: float

@app.post("/api/v1/decision", response_model=DecisionResponse)
async def get_trading_decision(
    request: MarketStateRequest,
    credentials: HTTPAuthorizationCredentials = Depends(security)
):
    """Get real-time trading decision from Forward service."""
    # Authenticate
    validate_token(credentials.credentials)

    # Convert to protobuf
    grpc_request = forward_service_pb2.DecisionRequest(
        timestamp=request.timestamp,
        price_series=request.price_series,
        volume_series=request.volume_series,
        # ... other fields
    )

    # Call Rust Forward service via gRPC
    start = asyncio.get_event_loop().time()
    try:
        grpc_response = await forward_stub.GetDecision(grpc_request
            )
    except grpc.RpcError as e:
        raise HTTPException(status_code=503, detail=f"Forward␣
            service␣unavailable:␣{e}")

```

```
    latency = (asyncio.get_event_loop().time() - start) * 1000

    return DecisionResponse(
        action=grpc_response.action,
        confidence=grpc_response.confidence,
        ltn_satisfaction=grpc_response.ltn_satisfaction,
        risk_score=grpc_response.risk_score,
        latency_ms=latency,
    )

@app.post("/api/v1/sleep-cycle")
async def trigger_sleep_cycle(background_tasks: BackgroundTasks):
    """Trigger memory consolidation (runs asynchronously)."""
    # Enqueue background job
    background_tasks.add_task(run_sleep_cycle_job)
    return {"status": "sleep_cycle_queued", "job_id": "unique-job-
        id"}

async def run_sleep_cycle_job():
    """Background task that calls Backward service."""
    grpc_request = backward_service_pb2.SleepCycleRequest()

    try:
        grpc_response = await backward_stub.RunSleepCycle(
            grpc_request)
        print(f"Sleep␣cycle␣completed:␣{grpc_response.num_schemas}␣
            schemas")
    except grpc.RpcError as e:
        print(f"Sleep␣cycle␣failed:␣{e}")

# Monitoring endpoints
@app.get("/api/v1/health")
async def health_check():
    """Health check endpoint."""
    return {
        "status": "healthy",
        "forward_service": await check_forward_health(),
        "backward_service": await check_backward_health(),
    }

@app.get("/api/v1/metrics")
```

```python
async def get_metrics():
    """Prometheus metrics endpoint."""
    # Aggregate metrics from Rust services
    forward_metrics = await forward_stub.GetMetrics(empty_pb2.Empty())
    backward_metrics = await backward_stub.GetMetrics(empty_pb2.Empty())

    return {
        "forward": forward_metrics,
        "backward": backward_metrics,
    }
```

5.3 Batch Job Processing with Celery

```python
# gateway/tasks.py
from celery import Celery
import grpc

celery_app = Celery('janus', broker='redis://localhost:6379/0')

@celery_app.task(bind=True, max_retries=3)
def run_sleep_cycle(self):
    """Celery task for long-running sleep cycle."""
    channel = grpc.insecure_channel('backward-service:50052')
    stub = backward_service_pb2_grpc.BackwardServiceStub(channel)

    try:
        request = backward_service_pb2.SleepCycleRequest()
        response = stub.RunSleepCycle(request)

        return {
            "status": "success",
            "num_schemas": response.num_schemas,
            "duration_seconds": response.duration_seconds,
        }
    except grpc.RpcError as e:
        # Retry on failure
        raise self.retry(exc=e, countdown=60)

@celery_app.task
```

```python
def backtest_strategy(start_date: str, end_date: str):
    """Long-running backtesting job."""
    # Call Backward service for historical simulation
    # This can take hours, so runs in background
    pass
```

6 Hybrid Training Pipeline

6.1 PyTorch Training (Phase 1)

```python
# training/train_vivit.py
import torch
import torch.nn as nn
from torch.utils.data import DataLoader

class ViViTTrainer:
    def __init__(self, model, device='cuda'):
        self.model = model.to(device)
        self.device = device
        self.optimizer = torch.optim.AdamW(model.parameters(), lr=1
            e-4)

    def train_epoch(self, dataloader):
        self.model.train()
        total_loss = 0

        for batch in dataloader:
            videos, labels = batch
            videos = videos.to(self.device)
            labels = labels.to(self.device)

            # Forward pass
            outputs = self.model(videos)
            loss = nn.CrossEntropyLoss()(outputs, labels)

            # Backward pass
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

```

```
            total_loss += loss.item()

        return total_loss / len(dataloader)

    def export_to_onnx(self, path: str):
        """Export trained model to ONNX for Rust inference."""
        self.model.eval()
        dummy_input = torch.randn(1, 16, 2, 60, 60).to(self.device)

        torch.onnx.export(
            self.model,
            dummy_input,
            path,
            export_params=True,
            opset_version=14,
            do_constant_folding=True,
            input_names=['video'],
            output_names=['embedding'],
            dynamic_axes={
                'video': {0: 'batch_size'},
                'embedding': {0: 'batch_size'}
            }
        )
        print(f"Model exported to {path}")

# Usage
trainer = ViViTTrainer(model)
for epoch in range(100):
    loss = trainer.train_epoch(train_loader)
    print(f"Epoch {epoch}: Loss = {loss}")

# Export to ONNX
trainer.export_to_onnx("models/vivit.onnx")
```

6.2 Model Export & Validation

```
# training/validate_export.py
import torch
import onnx
import onnxruntime as ort
import numpy as np
```

```python

def validate_onnx_export(pytorch_model, onnx_path):
    """Validate that ONNX export matches PyTorch output."""

    # Load ONNX model
    onnx_model = onnx.load(onnx_path)
    onnx.checker.check_model(onnx_model)

    # Create ONNX Runtime session
    ort_session = ort.InferenceSession(onnx_path)

    # Test input
    dummy_input = torch.randn(1, 16, 2, 60, 60)

    # PyTorch inference
    pytorch_model.eval()
    with torch.no_grad():
        pytorch_output = pytorch_model(dummy_input).numpy()

    # ONNX inference
    ort_inputs = {ort_session.get_inputs()[0].name: dummy_input.
       numpy()}
    onnx_output = ort_session.run(None, ort_inputs)[0]

    # Compare outputs
    diff = np.abs(pytorch_output - onnx_output).max()
    print(f"Max difference: {diff}")

    if diff < 1e-5:
        print("[OK] ONNX export validated successfully")
    else:
        print("[FAIL] ONNX export validation failed")

validate_onnx_export(model, "models/vivit.onnx")
```

7 Deployment Architecture

7.1 Docker Compose Setup

```yaml
# docker-compose.yml
```

```
version: '3.8'

services:
  gateway:
    build: ./gateway
    ports:
      - "8000:8000"
    environment:
      - FORWARD_SERVICE_URL=forward:50051
      - BACKWARD_SERVICE_URL=backward:50052
    depends_on:
      - forward
      - backward
      - redis

  forward:
    build: ./services/forward
    ports:
      - "50051:50051"
    volumes:
      - ./models:/models:ro
    environment:
      - RUST_LOG=info
      - MODEL_PATH=/models/vivit.onnx
    deploy:
      resources:
        limits:
          cpus: '4'
          memory: 2G

  backward:
    build: ./services/backward
    ports:
      - "50052:50052"
    volumes:
      - ./models:/models:ro
      - ./data:/data
    environment:
      - RUST_LOG=info
      - QDRANT_URL=http://qdrant:6333
    depends_on:
```

```
      - qdrant

  qdrant:
    image: qdrant/qdrant:latest
    ports:
      - "6333:6333"
    volumes:
      - qdrant_data:/qdrant/storage

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  celery_worker:
    build: ./gateway
    command: celery -A tasks worker --loglevel=info
    depends_on:
      - redis
      - backward

volumes:
  qdrant_data:
```

7.2 Kubernetes Deployment

```
# k8s/forward-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: janus-forward
spec:
  replicas: 3
  selector:
    matchLabels:
      app: janus-forward
  template:
    metadata:
      labels:
        app: janus-forward
    spec:
```

```
      containers:
      - name: forward
        image: janus/forward:latest
        ports:
        - containerPort: 50051
        resources:
          requests:
            memory: "1Gi"
            cpu: "2"
          limits:
            memory: "2Gi"
            cpu: "4"
        env:
        - name: RUST_LOG
          value: "info"
        - name: MODEL_PATH
          value: "/models/vivit.onnx"
        volumeMounts:
        - name: models
          mountPath: /models
          readOnly: true
      volumes:
      - name: models
        persistentVolumeClaim:
          claimName: model-storage
---
apiVersion: v1
kind: Service
metadata:
  name: janus-forward
spec:
  selector:
    app: janus-forward
  ports:
  - protocol: TCP
    port: 50051
    targetPort: 50051
  type: ClusterIP
```

8 Implementation Roadmap

8.1 Phase 1: Foundation (Weeks 1-4)

1. **Week 1: Project Setup**
   - Initialize Rust workspace with cargo
   - Set up CI/CD (GitHub Actions)
   - Create protobuf definitions
   - Generate gRPC stubs for Rust and Python
2. **Week 2: Core Crates**
   - Implement `janus-core` with domain types
   - Implement `janus-vision` GAF transformation
   - Add comprehensive unit tests
3. **Week 3: Model Integration**
   - Train ViViT model in PyTorch
   - Export to ONNX and validate
   - Integrate ONNX Runtime in Rust
   - Benchmark inference latency
4. **Week 4: Basic Services**
   - Implement Forward service skeleton
   - Implement Backward service skeleton
   - Set up FastAPI gateway
   - Test end-to-end flow

8.2 Phase 2: Core Features (Weeks 5-8)

1. **Week 5: LTN Implementation**
   - Implement Łukasiewicz t-norms
   - Encode wash sale constraint
   - Encode Almgren-Chriss constraint
   - Encode VPIN constraint

2. **Week 6: Decision Engine**
    - Implement dual pathway logic
    - Add risk gating
    - Add logic gating
    - Implement cerebellar forward model
3. **Week 7: Memory System**
    - Implement episodic buffer
    - Implement prioritized replay
    - Add schema consolidation
4. **Week 8: Vector Database**
    - Set up Qdrant
    - Implement schema storage
    - Implement similarity search
    - Add periodic pruning

8.3 Phase 3: Production Readiness (Weeks 9-12)

1. **Week 9: Performance Optimization**
    - Profile and optimize hot paths
    - Add SIMD optimizations
    - Implement model quantization
    - Benchmark end-to-end latency
2. **Week 10: Safety & Testing**
    - Remove all panic!() calls
    - Add comprehensive error handling
    - Write integration tests
    - Add property-based tests
3. **Week 11: Monitoring & Observability**
    - Add Prometheus metrics
    - Implement distributed tracing

- Set up alerting
- Create dashboards

4. **Week 12: Deployment**
   - Create Docker images
   - Write Kubernetes manifests
   - Set up staging environment
   - Run load tests

9 Complete Implementation Checklist

sec:checklist

9.1 Infrastructure Setup

- Create Rust workspace (`Cargo.toml`)
- Set up monorepo structure (services + crates)
- Configure CI/CD pipeline
- Set up Docker build system
- Configure linting (clippy, rustfmt)

9.2 Shared Crates

- **janus-core**: Domain types, errors, utilities
- **janus-vision**: GAF, ViViT, image processing
- **janus-logic**: LTN, constraint evaluation
- **janus-memory**: Replay buffer, schemas
- **janus-execution**: Order execution, market simulation
- **janus-proto**: Protobuf definitions and gRPC stubs

9.3 Forward Service (Rust)

- gRPC server setup with Tonic
- GAF transformation pipeline
- ONNX model loading and inference
- LTN constraint evaluation
- Multimodal fusion
- Decision engine with dual pathways
- Metrics export (Prometheus)
- Health check endpoint
- Graceful shutdown

9.4 Backward Service (Rust)

- gRPC server setup
- Episodic buffer implementation
- Prioritized replay sampling
- TD-error computation
- Schema consolidation
- UMAP integration
- Qdrant client and sync
- Sleep cycle orchestration
- Parallel batch processing

9.5 Gateway Service (Python)

- FastAPI application setup
- gRPC client stubs
- REST endpoints (/decision, /sleep-cycle, etc.)
- JWT authentication

- Rate limiting middleware
- Celery worker setup
- Background job management
- Health check aggregation
- Metrics aggregation

9.6 Training Pipeline (Python)

- PyTorch model definitions
- Data loaders and preprocessing
- Training loops
- ONNX export scripts
- Export validation
- Model versioning

9.7 Testing

- Unit tests for all crates
- Integration tests for services
- End-to-end tests
- Load tests
- Chaos testing

9.8 Deployment

- Docker images for all services
- Docker Compose for local dev
- Kubernetes manifests
- Helm charts
- Staging environment
- Production environment

10 Conclusion

This Rust-first implementation strategy for JANUS provides:

1. **Maximum Performance**: Rust's zero-cost abstractions and no GIL enable <100ms latency for Forward service
2. **Type Safety**: Compile-time guarantees prevent entire classes of bugs
3. **Memory Safety**: No null pointer dereferences, no data races
4. **Developer Experience**: Python gateway maintains ease of use for API consumers
5. **Ecosystem Compatibility**: Hybrid approach leverages best tools from both ecosystems

Next Steps:

- Begin with Phase 1 foundation (weeks 1-4)
- Train initial models in PyTorch and export to ONNX
- Implement Forward service with ONNX Runtime
- Gradually migrate components to pure Rust (Candle/Burn)
- Monitor performance and iterate

The architecture is designed for **incremental migration**, allowing you to start with ONNX inference in Rust while keeping training in PyTorch, then gradually move to a fully Rust-native stack as the ecosystem matures.

References

[1] Jordan Smith, "JANUS Forward: Wake State Logic Trading Algorithm," 2025.

[2] Jordan Smith, "JANUS Backward: Sleep State Memory Management," 2025.

[3] Laurent Mazare, "tch-rs: Rust bindings for PyTorch," GitHub, 2024.

[4] Microsoft, "ONNX Runtime: Cross-platform ML inference," 2024.

[5] Hugging Face, "Candle: Minimalist ML framework in Rust," 2024.

[6] Burn Contributors, "Burn: Deep learning framework in Rust," 2024.

[7] Tokio Contributors, "Tonic: A native gRPC client & server implementation," 2024.

[8] Sebastián Ramírez, "FastAPI: Modern, fast web framework for Python," 2024.

[9] Rayon Contributors, "Rayon: Data parallelism library for Rust," 2024.

[10] ndarray Contributors, "ndarray: N-dimensional arrays for Rust," 2024.

[11] Qdrant Team, "Qdrant: Vector database for ML applications," 2024.

End of Complete Documentation Suite

For updates, implementation code, and the latest PDFs, visit:

https://github.com/nuniesmith/technical_papers

Individual Documents Available:

- `main.pdf` — Main Architecture
- `forward.pdf` — Forward Service
- `backward.pdf` — Backward Service
- `neuro.pdf` — Neuromorphic Architecture
- `rust.pdf` — Rust Implementation

Project JANUS — Looking simultaneously to the future and the past