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 25, 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

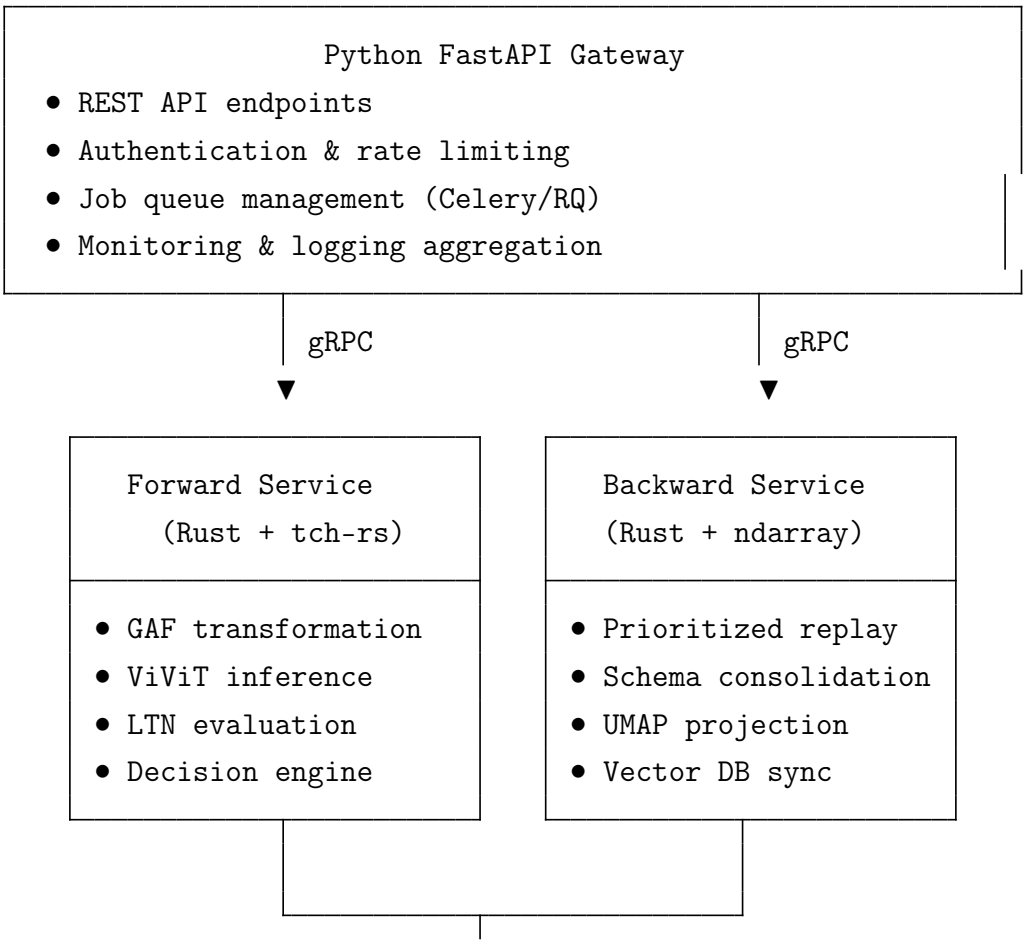
1.1 The Rust-First Philosophy

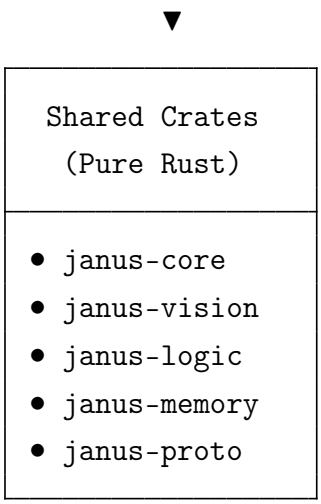
Traditional ML systems place Python at the center, with C++/Rust used only for performance-critical bottlenecks. JANUS inverts this model:

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;
```

```rust

        // Implementation of ukasiewicz t-norm
        for (t, past_action) in history.iter().enumerate() {
            if let Action::Sell { .. } = past_action {
                if is_loss(past_action) {
                    for k in 1..=30 {
                        if t + k < history.len() {
                            if let Action::Buy { .. } = &history[t + k] {
                                // ukasiewicz conjunction: max(0, a + b - 1)
                                let conjunction = (1.0 + 1.0 - 1.0).max(0.0);
                                // Negation: 1 - conjunction
                                let satisfaction = 1.0 - conjunction;
                                min_satisfaction = min_satisfaction.min(satisfaction);
                            }
                        }
                    }
                }
            }
        }

        min_satisfaction
    }

    pub fn check_almgren_chriss(&self, state: &MarketState, action: &Action) -> f32 {
        let volatility = state.volatility;
        let avg_volume = state.volume_series.mean().unwrap();

        match action {
            Action::Buy { size } | Action::Sell { size } => {
                let eta = 0.1;  // Market impact coefficient
                let threshold = eta * volatility * (size / avg_volume).sqrt();

                // If impact < threshold, satisfaction = 1.0
                // 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")
```

```
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

```
# 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

```
# 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.