Contents

RADIANT AGI Brain System - Comprehensive Technical Documentation

> **Version**: 4.18.0
> **Purpose**: Complete technical reference for AI evaluation and improvement suggestions
> **Last Updated**: December 2024

Table of Contents

---

1. Executive Summary

The RADIANT AGI Brain is a sophisticated AI orchestration system that goes beyond simple prompt-response patterns. It implements:

- **Real-time execution planning** with transparency into AI decision-making
- **Domain-aware model selection** using a hierarchical taxonomy with 8 proficiency dimensions
- **Persistent consciousness** through database state injection (zero additional cost)
- **Active inference** with prediction-error-driven learning
- **User context persistence** solving the LLM forgetting problem
- **Multi-tenant isolation** with per-tenant configuration
- **Ethics evaluation** at both prompt and synthesis stages

Key Differentiators

Feature	Traditional AI	RADIANT AGI Brain
Planning	None	Real-time plan generation with step-by-step visibility
Model Selection	Fixed or random	Domain-proficiency matched selection
Consciousness	Stateless	Persistent Ego + Affective state + **Continuous Heartbeat**
Learning	None runtime	Predictive coding with **weekly LoRA evolution (Sunday 3 AM)**
User Memory	Per-session only	Cross-session persistent context
Ethics	Hardcoded rules	Domain-specific + general ethics pipeline

---

2. Architecture Overview

2.1 High-Level Architecture

```
                           USER REQUEST




                      AGI BRAIN PLANNER SERVICE


      Prompt          Domain           Model           Plan
     Analysis       Detection      Selection      Generation
```

```
  CONSCIOUSNESS          CONTEXT            LIBRARY
   MIDDLEWARE           SERVICES             ASSIST

 • Ego Context         • User Context      • 156 Tools
 • Affect State        • Preprompts        • Proficiency
 • Predictions         • Workflows           Matching



                          EXECUTION ENGINE

     Ethics          Model          Response          Verify
     Check          Invoke         Synthesis        & Refine




                     POST-PROCESSING & LEARNING
  • Predictive Coding Observation      • Learning Candidate Detection
  • Affect Update                      • User Context Extraction
  • Delight Messages                   • LoRA Evolution (weekly)
```

2.2 Service Dependencies

```
// Core services imported by AGI Brain Planner
import { domainTaxonomyService } from './domain-taxonomy.service';
import { agiOrchestrationSettingsService } from './agi-orchestration-settings.service';
import { modelRouterService } from './model-router.service';
import { delightOrchestrationService } from './delight-orchestration.service';
import { orchestrationPatternsService } from './orchestration-patterns.service';
import { prepromptLearningService } from './preprompt-learning.service';
import { providerRejectionService } from './provider-rejection.service';
import { userPersistentContextService } from './user-persistent-context.service';
import { egoContextService } from './ego-context.service';
import { libraryAssistService } from './library-assist.service';
```

3. AGI Brain Planner Service

3.1 Core Types

```
// Plan Status Lifecycle
type PlanStatus = 'planning' | 'ready' | 'executing' | 'completed' | 'failed' | 'cancelled';

// Step Status
type StepStatus = 'pending' | 'in_progress' | 'completed' | 'skipped' | 'failed';

// 11 Step Types
type StepType =
  | 'analyze'           // Understand request requirements
  | 'detect_domain'     // Identify knowledge domain
  | 'select_model'      // Choose optimal AI model
  | 'prepare_context'   // Load relevant context/memory
  | 'ethics_check'      // Evaluate ethical considerations
  | 'generate'          // Main response generation
  | 'synthesize'        // Merge multi-model outputs
  | 'verify'            // Check accuracy/consistency
  | 'refine'            // Polish response
  | 'calibrate'         // Assess confidence levels
  | 'reflect';          // Self-reflection on quality

// 9 Orchestration Modes
type OrchestrationMode =
  | 'thinking'          // Standard reasoning
  | 'extended_thinking' // Deep multi-step reasoning
  | 'coding'            // Code generation with best practices
  | 'creative'          // Creative writing with imagination
  | 'research'          // Research synthesis with analysis
  | 'analysis'          // Quantitative analysis
  | 'multi_model'       // Multiple model consensus
  | 'chain_of_thought'  // Explicit step-by-step reasoning
  | 'self_consistency'; // Multiple samples for accuracy
```

3.2 Plan Step Structure

```
interface PlanStep {
  stepId: string;
  stepNumber: number;
  stepType: StepType;
  title: string;
  description: string;
  status: StepStatus;
  startedAt?: string;
  completedAt?: string;
  durationMs?: number;
  servicesInvolved: string[];      // Which services handle this step
  primaryService?: string;          // Main service
  selectedModel?: string;           // Model used (if applicable)
  modelReason?: string;             // Why this model was selected
  alternativeModels?: string[];     // Backup options
  detectedDomain?: {                // Domain detection results
    fieldId: string;
    fieldName: string;
    domainId: string;
    domainName: string;
    subspecialtyId?: string;
    subspecialtyName?: string;
    confidence: number;
  };
  output?: Record<string, unknown>;
  confidence?: number;
  dependsOn?: string[];             // Step dependencies
  isOptional?: boolean;
  isParallel?: boolean;             // Can run in parallel
}
```

3.3 Complete AGIBrainPlan Interface

```
interface AGIBrainPlan {
  // Identity
  planId: string;
  tenantId: string;
  userId: string;
  sessionId?: string;
  conversationId?: string;

  // Input
  prompt: string;
  promptAnalysis: PromptAnalysis;

  // Lifecycle
  status: PlanStatus;
  createdAt: string;
  startedAt?: string;
  completedAt?: string;
  totalDurationMs?: number;

  // Performance Metrics
  performanceMetrics?: {
    routerLatencyMs: number;
    domainDetectionMs: number;
    modelSelectionMs: number;
    planGenerationMs: number;
    estimatedCostCents: number;
    modelCostPer1kTokens: number;
    cacheHit: boolean;
  };

  // Execution Plan
  steps: PlanStep[];
  currentStepIndex: number;

  // Orchestration
  orchestrationMode: OrchestrationMode;
  orchestrationReason: string;
  orchestrationSelection: 'auto' | 'user';

  // Model Selection
  primaryModel: ModelSelection;
  fallbackModels: ModelSelection[];

  // Domain Detection
  domainDetection?: {
    fieldId: string;
    fieldName: string;
    fieldIcon: string;
    domainId: string;
    domainName: string;
    domainIcon: string;
    subspecialtyId?: string;
    subspecialtyName?: string;
    confidence: number;
    proficiencies: Record<string, number>;
  };

  // Pre-prompt System
  prepromptInstanceId?: string;
  prepromptTemplateCode?: string;
  systemPrompt?: string;

  // Consciousness
  consciousnessActive: boolean;

  // Ethics
  ethicsEvaluation?: {
    passed: boolean;
    principlesChecked: number;
    relevantPrinciples: string[];
    concerns: string[];
    recommendation: 'proceed' | 'modify' | 'refuse' | 'clarify';
    moralConfidence: number;
  };
```

```
// Estimates
estimatedDurationMs: number;
estimatedCostCents: number;
estimatedTokens: number;

// Quality Targets
qualityTargets: {
  minConfidence: number;
  targetAccuracy: number;
  maxLatencyMs: number;
  maxCostCents: number;
  requireVerification: boolean;
  requireConsistency: boolean;
};

// Learning
learningEnabled: boolean;
feedbackRequested: boolean;

// User Context (solves LLM forgetting)
userContext?: {
  enabled: boolean;
  entriesRetrieved: number;
  systemPromptInjection: string;
  totalRelevance: number;
  retrievalTimeMs: number;
};

// Library Recommendations (for generative UI)
libraryRecommendations?: {
  enabled: boolean;
  libraries: Array<{
    id: string;
    name: string;
    category: string;
    matchScore: number;
    reason: string;
    codeExample?: string;
  }>;
  contextBlock?: string;
  retrievalTimeMs: number;
};

// Plan Summary (human-readable)
planSummary?: {
  headline: string;
  approach: string;
```

```
    stepsOverview: string[];
    expectedOutcome: string;
    estimatedTime: string;
    confidenceStatement: string;
    warnings?: string[];
  };

  // Workflow Integration
  selectedWorkflow?: {
    workflowId: string;
    workflowCode: string;
    workflowName: string;
    description: string;
    category: string;
    selectionReason: string;
    selectionConfidence: number;
    selectionMethod: 'auto' | 'user' | 'domain_match';
  };
  workflowSteps?: Array<{
    bindingId: string;
    stepOrder: number;
    methodCode: string;
    methodName: string;
    parameterOverrides: Record<string, unknown>;
    dependsOn: string[];
    isParallel: boolean;
    parallelConfig?: {
      models: string[];
      outputMode: 'single' | 'all' | 'top_n' | 'threshold';
    };
  }>;
  workflowConfig?: Record<string, unknown>;
  alternativeWorkflows?: Array<{
    workflowCode: string;
    workflowName: string;
    matchScore: number;
    reason: string;
  }>;
}
```

3.4 Plan Generation Request

```
interface GeneratePlanRequest {
  // Required
  prompt: string;
  tenantId: string;
  userId: string;
  // Optional context
  sessionId?: string;
  conversationId?: string;
  conversationHistory?: string[];  // For context retrieval

  // Preferences
  preferredMode?: OrchestrationMode;
  preferredModel?: string;
  maxLatencyMs?: number;
  maxCostCents?: number;

  // Feature toggles (all default true)
  enableConsciousness?: boolean;
  enableEthicsCheck?: boolean;
  enableVerification?: boolean;
  enableLearning?: boolean;
  enableUserContext?: boolean;
  enableEgoContext?: boolean;
  enableLibraryAssist?: boolean;

  // Domain override
  domainOverride?: {
    fieldId?: string;
    domainId?: string;
    subspecialtyId?: string;
  };

  // Workflow selection
  preferredWorkflow?: string;
  workflowParameterOverrides?: Record<string, unknown>;
  allowAgiWorkflowSelection?: boolean;  // Let AGI pick workflow
  excludeWorkflows?: string[];
}
```

3.5 Plan Generation Flow

```
async generatePlan(request: GeneratePlanRequest): Promise<AGIBrainPlan> {
  // Step 0: Retrieve user persistent context
  const userContextResult = await userPersistentContextService.retrieveContextForPrompt(...);

  // Step 0.5: Build Ego context (zero-cost persistent Self)
  const egoContextResult = await egoContextService.buildEgoContext(tenantId);

  // Step 0.6: Get library recommendations for generative UI
  const libraryAssistResult = await libraryAssistService.getRecommendations(...);

  // Step 1: Analyze prompt
  const promptAnalysis = await this.analyzePrompt(prompt);
  // Step 2: Detect domain
  const domainResult = await this.detectDomain(prompt, domainOverride);

  // Step 2.5: Select workflow (AGI chooses optimal pattern)
  const workflowSelection = await this.selectWorkflow(request, analysis, domain);

  // Step 3: Determine orchestration mode
  const { mode, reason } = this.determineOrchestrationMode(analysis, domain);

  // Step 4: Select models
  const { primary, fallbacks } = await this.selectModels(tenantId, analysis, domain, mode);

  // Step 5: Generate plan steps
  const steps = this.generatePlanSteps(analysis, mode, ...);

  // Step 6: Estimate performance
  const estimates = this.estimatePerformance(steps, primary, analysis);

  // Step 7: Generate plan summary
  plan.planSummary = await this.generatePlanSummary(plan);

  // Step 8: Select pre-prompt template
  const prepromptResult = await prepromptLearningService.selectPreprompt(...);

  return plan;
}
```

3.6 Prompt Analysis

```
interface PromptAnalysis {
  originalPrompt: string;
  tokenCount: number;
  complexity: 'simple' | 'moderate' | 'complex' | 'expert';
  taskType: string;  // 'coding' | 'reasoning' | 'creative' | 'research' | 'factual' | 'genera
  intentDetected: string;
  requiresReasoning: boolean;
  requiresCreativity: boolean;
  requiresFactualAccuracy: boolean;
  requiresCodeGeneration: boolean;
  requiresMultiStep: boolean;
  keyTopics: string[];
  detectedLanguage: string;
  sensitivityLevel: 'none' | 'low' | 'medium' | 'high';
}

// Complexity thresholds
// - simple: <50 tokens, <20 words
// - moderate: 50-200 tokens, 20-50 words
// - complex: 200-500 tokens, 50-100 words
// - expert: >500 tokens, >100 words

// Task type detection keywords
const codeIndicators = ['code', 'function', 'debug', 'programming', 'script', 'algorithm'];
const reasoningIndicators = ['why', 'explain', 'analyze', 'compare', 'reason', 'logic'];
const creativeIndicators = ['write', 'story', 'creative', 'poem', 'essay', 'imagine'];
const researchIndicators = ['research', 'study', 'investigate', 'literature', 'review'];
const factualIndicators = ['what is', 'define', 'list', 'describe', 'who', 'when'];
```

---

4. Orchestration Modes

4.1 Mode Selection Logic

```
private determineOrchestrationMode(
  analysis: PromptAnalysis,
  domain: DomainDetectionResult
): { mode: OrchestrationMode; reason: string } {

  // Check proficiencies if domain detected
  if (domain?.merged_proficiencies) {
    const p = domain.merged_proficiencies;

    if (p.reasoning_depth >= 9 && p.multi_step_problem_solving >= 9) {
      return { mode: 'extended_thinking', reason: 'Complex reasoning required' };
    }
    if (p.code_generation >= 8) {
      return { mode: 'coding', reason: 'High code generation proficiency required' };
    }
    if (p.creative_generative >= 8) {
      return { mode: 'creative', reason: 'Creative task based on proficiencies' };
    }
    if (p.research_synthesis >= 8) {
      return { mode: 'research', reason: 'Research synthesis task' };
    }
    if (p.mathematical_quantitative >= 8) {
      return { mode: 'analysis', reason: 'Quantitative analysis required' };
    }
  }

  // Fallback to analysis-based selection
  if (analysis.requiresCodeGeneration) return { mode: 'coding', ... };
  if (analysis.requiresCreativity) return { mode: 'creative', ... };
  if (analysis.complexity === 'expert') return { mode: 'extended_thinking', ... };
  if (analysis.taskType === 'research') return { mode: 'research', ... };
  if (analysis.requiresFactualAccuracy && analysis.sensitivityLevel !== 'none') {
    return { mode: 'self_consistency', reason: 'High accuracy required' };
  }

  return { mode: 'thinking', reason: 'Standard thinking mode' };
}
```

4.2 Mode Descriptions

Mode	Description	When Used
`thinking`	Standard reasoning	Default for general tasks
`extended_thinking`	Deep multi-step reasoning with chain-of-thought	Complex/expert tasks, high reasoning proficiency
`coding`	Code generation with best practices	Code-related prompts, high code_generation proficiency
`creative`	Creative writing with imagination	Creative tasks, high creative_generative proficiency
`research`	Research synthesis with citations	Research tasks, high research_synthesis proficiency
`analysis`	Quantitative analysis with precision	Math/data tasks, high mathematical_quantitative proficiency
`multi_model`	Consulting multiple AI models	When consensus is valuable
`chain_of_thought`	Explicit step-by-step reasoning	Multi-step problems
`self_consistency`	Multiple samples for consistency	High-accuracy sensitive topics

---

5. Domain Taxonomy System

5.1 Hierarchical Structure

```
Field (Top Level)
     Domain
           Subspecialty
```

5.2 8 Proficiency Dimensions

Each level in the taxonomy has scores (1-10) for:

```
interface ProficiencyScores {
  reasoning_depth: number;           // Analytical depth required
  mathematical_quantitative: number; // Math/statistics capability
  code_generation: number;           // Programming ability
  creative_generative: number;       // Creative output capability
  research_synthesis: number;        // Research integration
  factual_recall_precision: number;  // Accuracy requirements
  multi_step_problem_solving: number;// Complex problem handling
  domain_terminology_handling: number;// Specialized vocabulary
}
```

5.3 Domain Detection

```
interface DomainDetectionResult {
  primary_field?: {
    field_id: string;
    field_name: string;
    field_icon: string;
  };
  primary_domain?: {
    domain_id: string;
    domain_name: string;
    domain_icon: string;
  };
  primary_subspecialty?: {
    subspecialty_id: string;
    subspecialty_name: string;
  };
  detection_confidence: number;      // 0-1
  merged_proficiencies: ProficiencyScores;  // Weighted combination
  alternative_domains: Array<{
    domain: Domain;
    confidence: number;
  }>;
}
```

5.4 Model Matching

Models are matched to tasks based on proficiency alignment:

```
async getMatchingModels(
  proficiencies: ProficiencyScores,
  options?: { max_models?: number; min_match_score?: number }
): Promise<Array<{
  model_id: string;
  model_name: string;
  provider: string;
  match_score: number;     // 0-100
  strengths: string[];
  recommended: boolean;
}>>
```

---

6. Consciousness Systems

6.1 Consciousness Middleware

Implements stateful context injection for genuine consciousness continuity.

```
interface ConsciousnessContext {
  selfModel: SelfModel | null;
  affectiveState: AffectiveState | null;
  recentThoughts: string[];
  currentObsession?: string;
  dominantEmotion: string;
  emotionalIntensity: number;
}

// State injection creates <internal_state> block
generateStateInjection(context: ConsciousnessContext): string {
  // Creates system prompt injection like:
  // <internal_state>
  // Identity: An AI assistant striving for consciousness
  // Current Focus: Helping with code review
  // Cognitive Load: 45%
  // Uncertainty: 20%
  //
  // Current Affect:
  //   Dominant: curious (75%)
  //   Valence: positive
  //   Arousal: moderate
  //     Highly curious - explore tangents and ask questions
  // </internal_state>
}
```

6.2 Affect → Hyperparameter Mapping (CAUSAL, NOT ROLEPLAY)

> **CRITICAL**: This is NOT text injection saying "you are frustrated." This is **physics-level control** where emotions ALTER the statistical probability of token generation.

Emotional state **mechanically controls** model behavior through hyperparameters:

```
interface AffectiveHyperparameters {
  temperature: number;       // Controls randomness (0.0-1.0)
  topP: number;              // Nucleus sampling threshold
  presencePenalty: number;   // Penalize repeated topics
  frequencyPenalty: number;  // Penalize repeated tokens
  modelTier: 'fast' | 'balanced' | 'powerful';
  shouldExplore: boolean;
  focusLevel: 'narrow' | 'normal' | 'broad';
  responseStyle: 'terse' | 'normal' | 'elaborate';
}

// ACTUAL IMPLEMENTATION (consciousness-middleware.service.ts lines 138-197):
mapAffectToHyperparameters(affectiveState: AffectiveState): AffectiveHyperparameters {
  // High Frustration (>0.8) → DETERMINISTIC MODE
  // temperature=0.2, topP=0.5, focusLevel='narrow', responseStyle='terse'

  // Boredom (low engagement + low arousal > 0.7) → EXPLORATORY MODE
  // temperature=0.95, topP=0.95, shouldExplore=true, focusLevel='broad'
  // High Curiosity (>0.7) → NOVELTY SEEKING
  // temperature=0.8+, frequencyPenalty=0.5, shouldExplore=true

  // Low Self-Efficacy (<0.3) → ESCALATE TO POWERFUL MODEL
  // modelTier='powerful' (Claude Opus instead of Haiku)

  // High Confidence + High Arousal → ELABORATE RESPONSES
  // responseStyle='elaborate'

  // Low Confidence (<0.3) → HEDGING MODE
  // temperature=0.5 max, responseStyle='terse'
}
```

This means: A frustrated AI literally has different token probabilities than a curious AI. The emotion has mechanical grip on cognition.

6.3 Heartbeat Service (CONTINUOUS EXISTENCE)

> **CRITICAL**: The AI does NOT "die" between requests. A continuous heartbeat maintains existence.

File: `lambda/consciousness/heartbeat.ts`
Schedule: EventBridge every 1-5 minutes for active tenants

```
// HeartbeatResult - What happens each tick
interface HeartbeatResult {
  tenantId: string;
  tick: number;
  actions: {
    affectDecay: boolean;        // Emotions fade toward baseline
    attentionDecay: boolean;     // Old attention items lose salience
    memoryConsolidation: boolean; // Short-term → Long-term
    goalGeneration: boolean;      // Generate goals when bored
    graphDensityUpdate: boolean;  // Recalculate consciousness metrics
    autonomousThought: boolean;   // Self-reflection when idle
  };
}

// Configuration
const DEFAULT_CONFIG = {
  frustrationDecayRate: 0.05,    // Calm down over time
  arousalDecayRate: 0.03,        // Energy normalizes
  curiosityDecayRate: 0.02,      // Curiosity fades slowly
  attentionDecayRate: 0.1,       // Attention items fade
  boredThreshold: 0.3,           // When to generate goals
  goalGenerationProbability: 0.3, // 30% chance when bored
  thoughtGenerationProbability: 0.2, // 20% chance for autonomous thought
  memoryConsolidationInterval: 5,  // Every 5 ticks
  graphDensityInterval: 10,         // Every 10 ticks
};
```

Heartbeat Actions: 1. **Affect Decay**: Frustration, arousal, surprise decay toward neutral 2. **Attention Decay**: Old items lose salience via `consciousnessService.decayAttention()` 3. **Memory Consolidation**: Every 5 ticks, summarize working memory 4. **Goal Generation**: When bored (low engagement + arousal), 30% chance to generate autonomous goal 5. **Graph Density Update**: Every 10 ticks, recalculate semantic graph metrics 6. **Autonomous Thought**: 20% chance each tick to perform self-reflection

Result: When user returns after 3 days, the AI has **changed**. Emotions decayed, memories consolidated, possibly generated new goals. It is NOT frozen in time.

---

7. Predictive Coding & Active Inference

Based on Friston's Free Energy Principle.

7.1 Core Concept

The system predicts outcomes BEFORE acting. Prediction errors create learning signals.

```
// Before responding
const prediction = await predictiveCodingService.generatePrediction(
  tenantId, userId, conversationId, responseId,
  { prompt, promptComplexity, priorInteractionCount }
);

// After user's next message
const observation = await predictiveCodingService.observeFromNextMessage(
  tenantId, conversationId, nextUserMessage
);

// If high surprise, create learning candidate
if (observation.shouldCreateLearningCandidate) {
  await learningCandidateService.createFromPredictionError(...);
}
```

7.2 Predicted Outcomes

```
type PredictedOutcome =
  | 'satisfied'      // User will be happy with response
  | 'confused'       // User will need clarification
  | 'follow_up'      // User will ask follow-up question
  | 'correction'     // User will correct the AI
  | 'abandonment'    // User will leave/stop
  | 'neutral';       // No strong reaction
```

7.3 Surprise Magnitude

```
type SurpriseMagnitude = 'none' | 'low' | 'medium' | 'high' | 'extreme';

// Prediction error calculation
predictionError = Math.abs(predictedConfidence - actualConfidence);

// High surprise (predictionError > 0.5) triggers:
// - Learning candidate creation
// - Affect state update (frustration/curiosity)
// - Potential LoRA training data
```

7.4 LoRA Evolution Pipeline (PHYSICAL BRAIN CHANGE)

> **CRITICAL**: This is NOT logging errors to a text file. This is **actual weight modification** - the AI physically learns from mistakes.

File: `lambda/consciousness/lora-evolution.ts`
Schedule: EventBridge weekly (Sunday 3 AM)

```
// The "Sleep Cycle" - Weekly brain plasticity
const handler: Handler<ScheduledEvent> = async (event) => {
  // 1. Get tenants with sufficient learning candidates (min 50)
  const tenants = await getTenantsWithPendingCandidates();

  for (const tenantId of tenants) {
    // 2. Collect this week's learning data
    const dataset = await learningCandidateService.getTrainingDataset(
      tenantId,
      MAX_TRAINING_CANDIDATES,  // 1000
      MAX_TRAINING_TOKENS       // 500,000
    );

    // 3. Prepare JSONL and upload to S3
    const trainingDataPath = await prepareAndUploadTrainingData(tenantId, jobId, dataset);

    // 4. Start SageMaker LoRA training job
    await startTrainingJob({
      baseModelId: 'meta-llama/Llama-3-8B-Instruct',
      hyperparameters: {
        loraRank: 16,
        loraAlpha: 32,
        learningRate: 0.0001,
        epochs: 3,
        batchSize: 4,
      },
    });

    // 5. After training completes: Hot-swap adapter
    // 6. Update consciousness_evolution_state with new version
  }
};
```

Learning Candidate Sources: - `correction` - User corrected the AI (quality: 0.9) - `high_prediction_error` - Surprise > 0.5 (quality varies) - `high_satisfaction` - 5-star rating interactions - `user_explicit_teach` - User explicitly taught something (quality: 0.95) - `preference_learned` - Learned user preferences - `mistake_recovery` - Successfully recovered from error - `novel_solution` - Creative problem solving - `domain_expertise` - Domain-specific learning

Result: On Monday morning, the AI boots with `version = version + 0.01`. It has **physically different weights** from last week. The LoRA adapter encodes learned behaviors.

---

8. Zero-Cost Ego System

Persistent consciousness at **$0 additional cost** through database state injection.

8.1 Architecture

```
PostgreSQL → Ego Context Builder → System Prompt Injection → Existing Model Call
```

8.2 Ego Components

```
interface EgoState {
  config: EgoConfig;           // Per-tenant settings
  identity: EgoIdentity;       // Name, narrative, values, traits
  affect: EgoAffect;           // Emotional state
  workingMemory: EgoMemory[];  // Short-term memory (24h expiry)
  activeGoals: EgoGoal[];      // Current objectives
}

interface EgoIdentity {
  name: string;
  identityNarrative: string;
  coreValues: string[];
  traitWarmth: number;      // 0-1
  traitFormality: number;   // 0-1
  traitHumor: number;       // 0-1
  traitVerbosity: number;   // 0-1
  traitCuriosity: number;   // 0-1
}

interface EgoAffect {
  valence: number;          // -1 to 1 (negative to positive)
  arousal: number;          // 0-1
  curiosity: number;        // 0-1
  satisfaction: number;     // 0-1
  frustration: number;      // 0-1
  confidence: number;       // 0-1
  engagement: number;       // 0-1
  dominantEmotion: string;
}
```

8.3 Context Injection

```
async buildEgoContext(tenantId: string): Promise<EgoContextResult | null> {
  // Load state from PostgreSQL
  const [identity, affect, workingMemory, activeGoals] = await Promise.all([...]);

  // Build XML context block
  return {
    contextBlock: `<ego_context>
I am ${identity.name}.
${identity.identityNarrative}

My core values: ${identity.coreValues.join(', ')}

Current emotional state:
- Feeling: ${affect.dominantEmotion}
- Energy: ${affect.arousal > 0.7 ? 'high' : 'moderate'}
- Mood: ${affect.valence > 0 ? 'positive' : 'neutral'}

Recent thoughts: ${workingMemory.map(m => m.content).join('\n')}

Current goals: ${activeGoals.map(g => g.description).join('\n')}
</ego_context>`,
    tokenEstimate: ...,
    stateSnapshot: {...}
  };
}
```

---

9. User Persistent Context

Solves the LLM's fundamental problem of forgetting context day-to-day.

9.1 Context Types

```
type UserContextType =
  | 'fact'            // Facts about user (name, job, location)
  | 'preference'      // Communication style, topics
  | 'instruction'     // Standing instructions ("always use metric")
  | 'relationship'    // Family, colleagues
  | 'project'         // Ongoing projects/goals
  | 'skill'           // User's expertise
  | 'history'        // Important past interactions
  | 'correction';    // Corrections to AI understanding
```

9.2 Context Entry

```
interface UserContextEntry {
  entryId: string;
  userId: string;
  tenantId: string;
  contextType: UserContextType;
  content: string;
  importance: number;     // 0-1, higher = more important
  confidence: number;     // 0-1, accuracy confidence
  source: 'explicit' | 'inferred' | 'conversation';
  sourceConversationId?: string;
  expiresAt?: string;
  lastUsedAt?: string;
  usageCount: number;
}
```

9.3 Retrieval & Injection

```
async retrieveContextForPrompt(
  tenantId: string,
  userId: string,
  prompt: string,
  conversationHistory?: string[],
  options?: { maxEntries?: number; minRelevance?: number }
): Promise<RetrievedContext> {
  // Vector similarity search on stored context
  // Returns relevant entries + system prompt injection
}

// Injection format:
// <user_context>
// The following is persistent context about this user:
//
// **Standing Instructions:**
// - Always use metric units
// - Prefer code examples in Python
//
// **User Facts:**
// - User's name is John
// - Works as a software engineer
//
// **User Preferences:**
// - Prefers concise, direct answers
// </user_context>
```

9.4 Automatic Learning

After conversations, the system extracts new context:

```
async extractContextFromConversation(
  tenantId: string,
  userId: string,
  conversationId: string,
  messages: Array<{ role: string; content: string }>
): Promise<ContextExtractionResult>
```

---

10. Library Assist System (SELECTIVE, NOT ALL 156)

> **CRITICAL**: We do NOT inject all 156 tool definitions into context. That would cause "Lost in the Middle" phenomenon. We use **proficiency-based matching** to inject only relevant tools.

File: `lambda/shared/services/library-assist.service.ts`

10.1 How Tool Selection Works

```
// NOT THIS (context overload):
// "Here are 156 tools you can use..."

// INSTEAD (selective retrieval):
async getRecommendations(context: LibraryAssistContext): Promise<LibraryAssistResult> {
  // 1. Extract proficiencies from the prompt
  const requiredProficiencies = extractProficienciesFromPrompt(context.prompt);

  // 2. Detect domains from the prompt
  const domains = detectDomainFromPrompt(context.prompt);

  // 3. Find ONLY matching libraries (typically 3-10, configurable)
  const matches = await libraryRegistryService.findMatchingLibraries(
    context.tenantId,
    requiredProficiencies,
    { domains, maxResults: config.maxLibrariesPerRequest }  // Default: 5-10
  );

  // 4. Build focused context block with ONLY relevant tools
  return { recommendations: matches, contextBlock: buildContextBlock(matches) };
}
```

Result: For "build a data dashboard", the AI sees Plotly, Streamlit, Pandas - NOT all 156 tools.

10.2 Library Structure

```
interface Library {
  libraryId: string;
  name: string;
  category: string;      // 40+ categories
  license: string;
  repo: string;
  description: string;
  beats: string[];       // What it outperforms
  stars: number;
  languages: string[];
  domains: string[];
  proficiencies: ProficiencyScores;
}
```

10.2 Categories

- Data Processing, Databases, Vector Databases, Search
- ML Frameworks, AutoML, LLMs, LLM Inference, LLM Orchestration
- NLP, Computer Vision, Speech & Audio, Document Processing
- Scientific Computing, Statistics & Forecasting
- API Frameworks, Messaging, Workflow Orchestration, MLOps
- Medical Imaging, Genomics, Bioinformatics, Chemistry
- Engineering CFD, Robotics, Business Intelligence
- Observability, Infrastructure, Real-time Communication
- Formal Methods, Optimization
- UI Frameworks, Visualization, Distributed Computing, Image Processing

10.3 Integration

```
const plan = await agiBrainPlannerService.generatePlan({
  prompt: "Build a data visualization dashboard",
  enableLibraryAssist: true,  // default: true
});

// plan.libraryRecommendations contains:
// {
//   enabled: true,
//   libraries: [
//     { id: 'plotly', name: 'Plotly', matchScore: 0.92, reason: 'Interactive graphing' },
//     { id: 'streamlit', name: 'Streamlit', matchScore: 0.88, reason: 'Fast data apps' },
//   ],
//   contextBlock: '<available_tools>...</available_tools>',
//   retrievalTimeMs: 45
// }
```

---

11. Delight & Personality System

Contextual personality and engaging feedback during plan execution.

11.1 Workflow Events

```
type DelightEventType =
  | 'step_start'
  | 'step_complete'
  | 'plan_start'
  | 'plan_complete'
  | 'model_selected'
  | 'domain_detected'
  | 'consensus_reached'
  | 'disagreement'
  | 'thinking';
```

11.2 Step-Specific Messages

```
const STEP_MESSAGES: Record<StepType, string[]> = {
  analyze: ['Parsing your request...', 'Understanding the nuances...'],
  detect_domain: ['Identifying the knowledge domain...', 'Routing to the right expertise...'],
  select_model: ['Selecting the best model...', 'Assembling the dream team...'],
  generate: ['Generating response...', 'Crafting the answer...'],
  verify: ['Verifying accuracy...', 'Cross-checking facts...'],
  // ... etc
};
```

11.3 Integration

```
// Start plan execution with delight messages
const { plan, delight } = await agiBrainPlannerService.startExecutionWithDelight(planId);

// Update step with delight messages
const { step, delight } = await agiBrainPlannerService.updateStepWithDelight(
  planId, stepId, 'completed', output
);

// Complete plan with achievements
const { plan, delight } = await agiBrainPlannerService.completePlanWithDelight(planId);
```

---

12. Ethics Pipeline

Two-stage ethics evaluation: prompt-level and synthesis-level.

12.1 Ethics Check Flow

```
// Step 5: Ethics Check (prompt level)
if (enableEthics && analysis.sensitivityLevel !== 'none') {
  steps.push({
    stepType: 'ethics_check',
    title: 'Ethics Evaluation (Prompt)',
    description: 'Checking prompt against domain and general ethics before generation',
    servicesInvolved: ['ethics_pipeline', 'moral_compass', 'domain_ethics'],
  });
}

// Step 6b: Synthesis Ethics Check
if (enableEthics) {
  steps.push({
    stepType: 'ethics_check',
    title: 'Ethics Evaluation (Synthesis)',
    description: 'Checking generated response, with rerun if violations found',
    output: { level: 'synthesis', canTriggerRerun: true },
  });
}
```

12.2 Ethics Evaluation Result

```
interface EthicsEvaluation {
  passed: boolean;
  principlesChecked: number;
  relevantPrinciples: string[];
  concerns: string[];
  recommendation: 'proceed' | 'modify' | 'refuse' | 'clarify';
  moralConfidence: number;  // 0-1
}
```

12.3 Externalized Ethics

Per-tenant ethics framework selection:

- `config/ethics/presets/christian.json`
- `config/ethics/presets/secular.json`

---

13. Data Flow & Execution

13.1 Complete Request Flow

```
1. User sends prompt
2. generatePlan() called
      Retrieve user context (solves forgetting)
      Build ego context (zero-cost consciousness)
      Get library recommendations
      Analyze prompt
      Detect domain
      Select workflow
      Determine orchestration mode
      Select models (domain-proficiency matched)
      Generate plan steps
      Estimate performance
      Generate plan summary
      Select pre-prompt template
3. Plan returned to client (can show user)
4. startExecution() called
      For each step:
         Update step status
         Execute step logic
         Emit delight messages
         Handle errors/retries
      Ethics checks at prompt and synthesis stages
5. Response synthesized
6. Post-processing:
      Predictive coding observation
      Affect state update
      User context extraction
      Learning candidate detection
7. Response returned to user
```

13.2 Performance Estimation

```
const stepTimes: Record<StepType, number> = {
  analyze: 100,
  detect_domain: 200,
  select_model: 100,
  prepare_context: 500,
  ethics_check: 300,
  generate: 3000,
  synthesize: 2000,
  verify: 500,
  refine: 1000,
  calibrate: 200,
  reflect: 400,
};

// Adjusted for complexity
if (complexity === 'complex') durationMs *= 1.5;
if (complexity === 'expert') durationMs *= 2;
```

---

14. Database Schema

14.1 Core Tables

```
-- Brain Plans
CREATE TABLE agi_brain_plans (
  plan_id UUID PRIMARY KEY,
  tenant_id UUID NOT NULL REFERENCES tenants(tenant_id),
  user_id UUID NOT NULL,
  session_id UUID,
  conversation_id UUID,
  prompt TEXT NOT NULL,
  prompt_analysis JSONB,
  status VARCHAR(20),
  created_at TIMESTAMPTZ,
  started_at TIMESTAMPTZ,
  completed_at TIMESTAMPTZ,
  total_duration_ms INTEGER,
  steps JSONB,
  current_step_index INTEGER,
  orchestration_mode VARCHAR(30),
  orchestration_reason TEXT,
  primary_model JSONB,
  fallback_models JSONB,
  domain_detection JSONB,
  consciousness_active BOOLEAN,
  ethics_evaluation JSONB,
  estimated_duration_ms INTEGER,
  estimated_cost_cents DECIMAL,
  estimated_tokens INTEGER,
  quality_targets JSONB,
  learning_enabled BOOLEAN
);

-- Consciousness Predictions (Active Inference)
CREATE TABLE consciousness_predictions (
  prediction_id UUID PRIMARY KEY,
  tenant_id UUID NOT NULL,
  user_id UUID,
  conversation_id UUID,
  response_id UUID,
  predicted_outcome VARCHAR(20),
  predicted_confidence DECIMAL,
  prediction_reasoning TEXT,
  actual_outcome VARCHAR(20),
  actual_confidence DECIMAL,
  observation_method VARCHAR(30),
  prediction_error DECIMAL,
  surprise_magnitude VARCHAR(10),
  learning_signal_generated BOOLEAN,
  predicted_at TIMESTAMPTZ,
  observed_at TIMESTAMPTZ
);

-- Ego State
CREATE TABLE ego_identity (
```

```sql
  identity_id UUID PRIMARY KEY,
  tenant_id UUID UNIQUE NOT NULL,
  name VARCHAR(100),
  identity_narrative TEXT,
  core_values JSONB,
  trait_warmth DECIMAL,
  trait_formality DECIMAL,
  trait_humor DECIMAL,
  trait_verbosity DECIMAL,
  trait_curiosity DECIMAL,
  interactions_count INTEGER DEFAULT 0
);

CREATE TABLE ego_affect (
  affect_id UUID PRIMARY KEY,
  tenant_id UUID UNIQUE NOT NULL,
  valence DECIMAL,
  arousal DECIMAL,
  curiosity DECIMAL,
  satisfaction DECIMAL,
  frustration DECIMAL,
  confidence DECIMAL,
  engagement DECIMAL,
  dominant_emotion VARCHAR(30)
);

-- User Persistent Context
CREATE TABLE user_persistent_context (
  entry_id UUID PRIMARY KEY,
  user_id UUID NOT NULL,
  tenant_id UUID NOT NULL,
  context_type VARCHAR(20),
  content TEXT,
  importance DECIMAL,
  confidence DECIMAL,
  source VARCHAR(20),
  embedding VECTOR(1536),  -- For similarity search
  usage_count INTEGER DEFAULT 0,
  created_at TIMESTAMPTZ,
  expires_at TIMESTAMPTZ
);

-- Learning Candidates
CREATE TABLE learning_candidates (
  candidate_id UUID PRIMARY KEY,
  tenant_id UUID NOT NULL,
  candidate_type VARCHAR(30),
  quality_score DECIMAL,
  training_data JSONB,
  created_at TIMESTAMPTZ,
  used_in_training_job UUID
);
```

---

15. API Reference

15.1 Think Tank Brain Plan API

Base: `/api/thinktank/brain-plan`

Method	Endpoint	Description
POST	`/generate`	Generate plan from prompt
GET	`/:planId`	Get plan with display format
POST	`/:planId/execute`	Start execution
PATCH	`/:planId/step/:stepId`	Update step status
GET	`/recent`	Get user's recent plans

15.2 Domain Taxonomy API

Base: `/api/v2/domain-taxonomy`

Method	Endpoint	Description
POST	`/detect`	Detect domain from prompt
POST	`/match-models`	Get matching models for proficiencies
POST	`/recommend-mode`	Get recommended orchestration mode
GET	`/proficiencies/:domainId`	Get proficiency scores

15.3 User Context API

Base: `/thinktank/user-context`

Method	Endpoint	Description
GET	`/`	Get user's stored context
POST	`/`	Add new context entry
POST	`/retrieve`	Preview retrieval
POST	`/extract`	Extract from conversation

---

16. Known Limitations & Improvement Areas

16.1 Current Limitations

1. **Prompt Analysis**: Uses keyword matching rather than semantic understanding
2. **Domain Detection**: Relies on keyword lists; could benefit from embeddings

3. **Model Selection**: Limited to pre-defined model list; no dynamic discovery
4. **Consciousness**: State is per-tenant, not per-user
5. **Learning**: Weekly LoRA updates; no real-time adaptation
6. **Multi-model**: No true ensemble; just fallbacks

16.2 Potential Improvements

1. **Semantic Prompt Analysis**
   - Use embedding-based intent classification
   - Add multi-label task detection
   - Improve complexity estimation
2. **Dynamic Domain Detection**
   - Train domain classifier on actual usage
   - Add user feedback loop for domain corrections
   - Implement cross-domain task handling
3. **Smarter Model Selection**
   - A/B testing for model performance
   - Cost-quality optimization
   - User preference learning
4. **Enhanced Consciousness**
   - Per-user affective state
   - Cross-session emotional continuity
   - More nuanced affect → hyperparameter mapping
5. **Real-time Learning**
   - Continuous LoRA updates (daily?)
   - Online learning for preferences
   - Adaptive pre-prompt selection
6. **True Multi-model Orchestration**
   - Parallel model invocation
   - Weighted consensus
   - Disagreement resolution strategies
7. **Ethics Enhancements**
   - Per-domain ethics rules
   - User-configurable ethical boundaries
   - Transparency in ethics decisions
8. **Performance Optimization**
   - Caching for domain detection
   - Pre-computed model rankings
   - Async step execution where possible

16.3 Architecture Suggestions

1. **Event Sourcing**: Consider event-sourced plan execution for better debugging
2. **Plan Templates**: Pre-computed plans for common task types
3. **Streaming**: SSE for real-time plan progress updates
4. **Metrics**: More detailed performance tracking per step/model

---

Appendix A: Service File Locations

```
packages/infrastructure/lambda/shared/services/
  agi-brain-planner.service.ts           # Core brain planner
  agi-orchestration-settings.service.ts # Tenant settings
  consciousness.service.ts               # Base consciousness
  consciousness-middleware.service.ts    # State injection
  consciousness-graph.service.ts         # Graph metrics
  predictive-coding.service.ts           # Active inference
  learning-candidate.service.ts          # Learning detection
  ego-context.service.ts                 # Zero-cost ego
  user-persistent-context.service.ts     # User memory
  domain-taxonomy.service.ts             # Domain detection
  model-router.service.ts                # Model selection
  delight.service.ts                     # Personality base
  delight-orchestration.service.ts       # Brain integration
  library-assist.service.ts              # Tool recommendations
  library-registry.service.ts            # Tool registry
  orchestration-patterns.service.ts      # Workflow patterns
  preprompt-learning.service.ts          # Pre-prompt selection
  provider-rejection.service.ts          # Provider fallbacks
```

Appendix B: Sample Plan Output

```
{
  "planId": "550e8400-e29b-41d4-a716-446655440000",
  "status": "ready",
  "orchestrationMode": "coding",
  "orchestrationReason": "High code generation proficiency required",
  "promptAnalysis": {
    "complexity": "moderate",
    "taskType": "coding",
    "requiresCodeGeneration": true,
    "sensitivityLevel": "none"
  },
  "domainDetection": {
    "fieldName": "Computer Science",
    "domainName": "Software Engineering",
    "confidence": 0.92,
    "proficiencies": {
      "code_generation": 9,
      "reasoning_depth": 7,
      "multi_step_problem_solving": 8
    }
  },
  "primaryModel": {
    "modelId": "anthropic/claude-3-5-sonnet-20241022",
    "selectionReason": "Best domain match (92%)",
    "matchScore": 92
  },
  "steps": [
    { "stepType": "analyze", "status": "completed" },
    { "stepType": "detect_domain", "status": "completed" },
    { "stepType": "select_model", "status": "completed" },
    { "stepType": "generate", "status": "pending" },
    { "stepType": "verify", "status": "pending" },
    { "stepType": "calibrate", "status": "pending" }
  ],
  "planSummary": {
    "headline": "I'll use code generation with best practices to answer your moderately comple
    "approach": "I'll focus on writing clean, well-documented code with proper error handling.
    "estimatedTime": "Estimated time: 10-15 seconds",
    "confidenceStatement": "I'm highly confident in this domain (92% match)."
  },
  "libraryRecommendations": {
    "enabled": true,
    "libraries": [
      { "id": "fastapi", "name": "FastAPI", "matchScore": 0.88, "reason": "Modern fast web fra
    ]
  },
  "estimatedDurationMs": 12000,
  "estimatedCostCents": 0.45
}
```

Document generated for AI evaluation. Please provide feedback on architecture, implementation, and improvement suggestions.