Neon-Affect

A CoJACK-Enabled Emotional Behavior Selection System

A Deep Code Analysis

Detailed Examination of the GetBehavior Pipeline, CoJACK Reservoir Dynamics, Level Cursor Triggering, OpenAI Integration, and CoJACK Behavior Selection

GitHub Repository: tenbase2-com-LLC / Neon-Affect

Repository Created: March 12, 2026

Analysis Date: March 13, 2026

1. Introduction

Neon-Affect is a CoJACK-enabled Java application built on the JACK BDI (Belief-Desire-Intention) agent framework with CoJACK cognitive extensions. The system processes **155 distinct emotions** mapped to **3,200 behaviors**, selecting context-appropriate responses by combining CoJACK's probabilistic plan selection with OpenAI's large language model capabilities and MBTI personality modelling. The project was created on March 12, 2026 and is described as still in active development.

The central architectural innovation is the pairing of a continuously-decaying emotional intensity reservoir (the NeonAffectReservoir moderator) with a two-pass OpenAI consultation: first to determine the *interaction modality* (verbal, nonverbal, or action), then to rank candidate behavior plans by likelihood given the agent's MBTI personality type and emotional history. The winning plan is then injected into CoJACK's expected-gain competition through dynamic formula updates, ensuring that CoJACK itself makes the final selection — preserving the cognitive plausibility of the BDI model.

1.1 Entry Point

The application entry point is **Program.java**:

```
import NeonAffect.NeonAffectAgent;
import NeonAffect.AffectEnum;

public class Program {
    public static void main( String args[] ) {
        try {
            long begTime = System.currentTimeMillis();

            NeonAffectAgent neonAgent = new NeonAffectAgent();
            neonAgent.PostInit();
            neonAgent.AddEmotionDirect(AffectEnum.RAGE);    // inject RAGE stimulus
            neonAgent.Wait(10.0);                           // let system run 10 seconds

            long endTime    = System.currentTimeMillis();
            long elapsedTime = endTime - begTime;
            System.out.println("\nElapsed time: " + elapsedTime);
            System.in.read();
        }
        catch (Exception e) { e.printStackTrace(); }
        System.exit(0);
    }
}
```

Three calls drive the entire pipeline: **PostInit()** starts all 155 emotion agents and their capabilities; **AddEmotionDirect(AffectEnum.RAGE)** injects the initial emotional stimulus directly into the CoJACK reservoir; and **Wait(10.0)** allows the asynchronous BDI plans to run to completion.

1.2 System-Level Architecture

The system consists of five interlocking layers:

- **CoJACK Cognitive Engine** — moderates reservoir values, computes plan activation and expected gain.
- **155 Emotion Agents** — each emotion (RAGE, GRIEF, AWE, …) has a dedicated NeonXxxAgent + XxxAffectCapability pair.

• **Level Cursor Subsystem** — 12 concurrent cursors map continuous reservoir values to discrete intensity levels.

• **OpenAI Bridge** — two separate GPT calls per behavior cycle determine modality and plan ranking.

• **Plan Context / Formula Injection** — selected plans are promoted via dynamic PLAN_VALUE updates before CoJACK's final competition.

2. Configuration Files

2.1 cogProcess.xml — Agent and Event Registry

The file **cogProcess.xml** is the JACK process descriptor. It lists every agent class, every event class, and every beliefset that JACK must load at startup. It registers all 155 emotion agent/capability pairs:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="agents">
    NeonAffect/NeonAffectAgent;NeonAffect/NeonAffectCapability;
    NeonAffect/AbstractMiniAgent;NeonAffect/AbstractAffectCapability;
    NeonAffect/NeonAnticipationAgent;NeonAffect/AnticipationAffectCapability;
    NeonAffect/NeonPleasedAgent;NeonAffect/PleasedAffectCapability;
    ... [155 emotion agent/capability pairs total] ...
    NeonAffect/NeonTimidAgent;NeonAffect/TimidAffectCapability
  </entry>
  <entry key="events">
    NeonAffect/AddEmotionEvent;
    NeonAffect/Level1PosAffectEvent;NeonAffect/Level2PosAffectEvent;
    NeonAffect/Level3PosAffectEvent;NeonAffect/Level4PosAffectEvent;
    NeonAffect/Level5PosAffectEvent;NeonAffect/Level6PosAffectEvent;
    NeonAffect/Level1NegAffectEvent;NeonAffect/Level2NegAffectEvent;
    NeonAffect/Level3NegAffectEvent;NeonAffect/Level4NegAffectEvent;
    NeonAffect/Level5NegAffectEvent;NeonAffect/Level6NegAffectEvent
  </entry>
  <entry key="beliefs">
    NeonAffect/Emotion
  </entry>
</properties>
```

2.2 cojack.xml — CoJACK Runtime Configuration

The file **data/cojack.xml** wires the CoJACK runtime to its supporting XML files and GUI windows:

```
<?xml version="1.0" encoding="UTF-8"?>
<properties>
  <entry key="aos.cojack.configuration">/data/cojack_cognitive.xml</entry>
  <entry key="aos.cojack.tracer_configuration">/data/cojack_tracer.xml</entry>
  <entry key="aos.cojack.datalog_configuration">/data/cojack_datalog.xml</entry>
  <entry key="aos.cojack.datalog_logfile">cojack.datalog.out</entry>
  <entry key="aos.cojack.configuration.trace_history_size">20</entry>
  <entry key="aos.cojack.tracer.traceOnGUI">true</entry>
  <entry key="aos.cojack.datalog.traceOnGUI">true</entry>
  <!-- Tracer GUI -->
  <entry key="aos.cojack.tracer.gui.title_window">CoJACK Trace</entry>
  <entry key="aos.cojack.tracer.gui.width">800</entry>
```

```
  <entry key="aos.cojack.tracer.gui.height">600</entry>
  <!-- Datalog GUI -->
  <entry key="aos.cojack.datalog.gui.title_window">CoJACK Datalog</entry>
  <entry key="aos.cojack.datalog.gui.width">800</entry>
  <entry key="aos.cojack.datalog.gui.height">600</entry>
</properties>
```

2.3 cojack_datalog.xml — Logging Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<datalogs maxTraceLineRecord="100">
    <datalog>
        <plot x="0" y="8"/>
        <key agent="*" class="PlanHistory"/>
    </datalog>
    <datalog>
        <plot x="0" y="1"/>
        <key agent="*" class="ModeratorLevel"/>
    </datalog>
</datalogs>
```

Two data streams are logged: **PlanHistory** records which plans were selected over time for every agent; **ModeratorLevel** records the reservoir value at each time step, allowing the emotional intensity trajectory to be visualised in the CoJACK Datalog GUI window.

3. The Initial Stimulus — AddEmotionDirect and AddEmotionPlan

When **neonAgent.AddEmotionDirect(AffectEnum.RAGE)** is called, the NeonAffectAgent posts an **AddEmotionEvent** directly into the JACK event queue. The event carries the moderator name (the string key for the CoJACK reservoir), the numeric amount to add, and a boolean *bIncremental* flag that determines whether the value is added to the existing reservoir or replaces it entirely.

The plan that handles this event is **AddEmotionPlan.java**:

```
public plan AddEmotionPlan extends Plan {
    #handles event AddEmotionEvent aee;
    #modifies data Emotion emotion;

    body() {
        String strModerator = aee.strModerator;   // e.g. "NeonAffectReservoir"
        double dValue       = aee.dAmount;         // e.g. -60.0 for RAGE
        long   lTime        = agent.timer.getTime();

        // Step 1: read the CoJACK reservoir value at the current simulation time
        double dReservoir = Configuration.moderatorCurrentReservoir(
                                agent.getId(), strModerator, lTime);

        // Step 2: combine with the incoming stimulus
        if (aee.bIncremental) {
            dReservoir = dReservoir + dValue;   // additive — most common case
        } else {
            dReservoir = dValue;                // absolute override
        }

        // Step 3: write the new value back into the CoJACK moderator
```

```
            Configuration.moderatorReservoirAgentUpdate(
                    agent.getId(), strModerator, dReservoir, lTime);

            // Step 4: re-read to confirm (CoJACK may clamp or adjust)
            dReservoir = Configuration.moderatorCurrentReservoir(
                            agent.getId(), strModerator, lTime);

            // Step 5: signal that the update is complete
            // (mutex4.signal() — unblocks the caller)
        }
    }
```

The two CoJACK API calls are critical. **Configuration.moderatorCurrentReservoir()** computes the reservoir value at time *lTime* by applying the decay formula defined in cojack_cognitive.xml from the last known value. This ensures the reservoir has already been decayed to the present moment before the new stimulus is applied — preventing the stimulus from being double-counted against a stale value.

Configuration.moderatorReservoirAgentUpdate() stores the new reservoir value and its timestamp, which becomes the baseline for all future decay calculations.

4. The CoJACK Moderated Reservoir

The emotional intensity of the agent at any moment is encoded in a single continuous value called the **reservoir** — referred to in the system as the **NeonAffectReservoir** moderator. This value is the heart of the entire system. All level cursor triggering, all OpenAI prompts, all plan selections ultimately flow from the current reservoir value.

4.1 The Reservoir Formula in cojack_cognitive.xml

The master reservoir formula is defined in the **NeonAffectGroup** parameter group inside **data/cojack_cognitive.xml**. It is evaluated automatically by the CoJACK engine at every simulation time step:

```
<parameter_group name="NeonAffectGroup">
    <param name="mbti_personality_type" value="INTP"/>
    <param name="neon_parameter" value="1"/>
    <moderator
        events_decay_time="0.0"
        reservoir_formula="if ((reservoir - 0.2525 * dt > 0) &amp;&amp; reservoir > 0.05) then
            (reservoir - 0.2525 * dt)
        else if ((reservoir + 0.2525 * dt &lt; 0) &amp;&amp; reservoir &lt; -0.05)
            then (reservoir + 0.2525 * dt)
        else 0.0;"
        type="NeonAffectReservoir">
        <param name="reservoir" value="0"/>
        <applies_to formula="reservoir" name="neon_parameter"/>
    </moderator>
</parameter_group>
```

4.2 Formula Analysis

The formula implements a **dead-band decay** toward zero with three branches:

Condition	Action	Meaning

`(reservoir - 0.2525*dt > 0) AND reservoir > 0.05`	`reservoir = reservoir - 0.2525`	dPtositive decay: emotion above +0 05 decays downward at 0.2525 units/second
`(reservoir + 0.2525*dt < 0) AND reservoir < -0.05`	`reservoir = reservoir + 0.2525`	dNtegative decay: emotion below -0 05 decays upward at 0.2525 units/second
`Otherwise`	`reservoir = 0.0`	Dead-band: small values snap to zero, preventing oscillation

The decay constant **0.2525 units/second** means a reservoir value of -60.0 (deep RAGE) will return to zero in approximately 237.6 seconds (~3.96 minutes) if no additional stimuli are added. The dead-band of ±0.05 prevents the reservoir from oscillating around zero due to floating-point imprecision.

4.3 GrabReservoirValuePlan — Polling the Reservoir

A dedicated polling plan runs continuously, reading the CoJACK reservoir value every 500 milliseconds and writing it to the agent's internal state so that level cursors can monitor it:

```
public plan GrabReservoirValuePlan extends Plan {
    #handles event GrabReservoirValueEvent grve;
    #uses interface NeonAffectAgent enc;

    body() {
        while(true) {
            @wait_for(elapsedMillis(TIME_OUT));  // wait 500 ms
            grabValue();
        }
    }

    void grabValue() {
        double v = enc.GetReservoirValue();   // reads CoJACK moderator
        enc.SetEmotionValue(v);               // writes to EmotionCursorVar
    }

    private static long TIME_OUT = 500;  // milliseconds
}
```

4.4 MapAffectPlan — Mapping Stimuli to Affect Levels

```
public plan MapAffectPlan extends Plan {
    #handles event MapAffectEvent mae;

    body() {
        int    iAffect     = mae.iAffect;
        double dEmotion    = enc.GetReservoirValue();
        double dNewEmotion = dEmotion + mae.dAmount;

        // Classify the new emotion intensity into a discrete level (-6 to +6)
        int iLevel    = LevelClassifier.GetLevel(dNewEmotion);

        // Look up what emotion type corresponds to this timeline + level
        int iNewAffect = enc.GetAffectFromTimeline(iAffect, iLevel);

        @subtask(saee.post(iNewAffect));        // set affect type beliefset
        enc.SetMappedAffect(iNewAffect);        // store for downstream use
    }
}
```

5. Level Cursor Triggering

The continuous reservoir value is discretised into 12 levels (−6 through +6, excluding 0) by a set of **CoJACK Change cursor objects**. Each cursor monitors the **EmotionCursorVar** and fires its associated plan when the value enters its band.

5.1 Level Ranges

Each level occupies a 10-unit band. The level boundaries are defined as static constants on each cursor class. The complete mapping is:

Level	Min Value	Max Value	Condition	Emotional Quality
+6	55.0	65.0	`v >= 55.0 && v < 65.0`	Extreme positive
+5	45.0	55.0	`v >= 45.0 && v < 55.0`	Very strong positive
+4	35.0	45.0	`v >= 35.0 && v < 45.0`	Strong positive
+3	25.0	35.0	`v >= 25.0 && v < 35.0`	Notable positive
+2	15.0	25.0	`v >= 15.0 && v < 25.0`	Moderate positive
+1	5.0	15.0	`v >= 5.0 && v < 15.0`	Mild positive
-1	-15.0	-5.0	`v > -15.0 && v <= -5.0`	Mild negative
-2	-25.0	-15.0	`v > -25.0 && v <= -15.0`	Moderate negative
-3	-35.0	-25.0	`v > -35.0 && v <= -25.0`	Notable negative
-4	-45.0	-35.0	`v > -45.0 && v <= -35.0`	Strong negative
-5	-55.0	-45.0	`v > -55.0 && v <= -45.0`	Very strong negative
-6	-65.0	-55.0	`v > -65.0 && v <= -55.0`	Extreme negative (RAGE)

5.2 Level Cursor Classes

Each cursor class extends CoJACK's **Change** class and overrides **condition()**:

```
// Level1PosAffect.java — triggers at mild positive intensity
public class Level1PosAffect extends Change {
    private EmotionCursorVar c;
    private String strCursorType = CursorEnum.LEVEL_CURSOR_POS_1;
    private int    iLevel        = CursorEnum.iLEVEL_CURSOR_POS_1;

    public final static double MAX_VALUE = 15.0;
    public final static double MIN_VALUE =  5.0;

    public boolean condition() {
        double dValue = c.GetValue();
        if (dValue >= MIN_VALUE && dValue < MAX_VALUE) return true;
        agent1.SetCursorAck(strCursorType, iLevel, false);  // not in range
        return false;
    }

    // Static version used by LevelClassifier
    public static boolean condition(double dValue) {
        return (dValue >= MIN_VALUE && dValue < MAX_VALUE);
```

```
    }
}

// Level6NegAffect.java — triggers at extreme negative intensity (RAGE territory)
public class Level6NegAffect extends Change {
    public final static double MAX_VALUE = -55.0;   // upper bound (less negative)
    public final static double MIN_VALUE = -65.0;   // lower bound (more negative)

    public boolean condition() {
        double dValue = c.GetValue();
        return (dValue > MIN_VALUE && dValue <= MAX_VALUE);
    }
    public static boolean condition(double dValue) {
        return (dValue > MIN_VALUE && dValue <= MAX_VALUE);
    }
}
```

5.3 LevelClassifier — Mapping a Value to a Level Integer

```
public class LevelClassifier {
    public static int GetLevel(double dValue) {
        if     (Level6NegAffect.condition(dValue))  return  -6;
        else if (Level5NegAffect.condition(dValue)) return -5;
        else if (Level4NegAffect.condition(dValue)) return -4;
        else if (Level3NegAffect.condition(dValue)) return -3;
        else if (Level2NegAffect.condition(dValue)) return -2;
        else if (Level1NegAffect.condition(dValue)) return -1;
        else if (Level1PosAffect.condition(dValue))  return  1;
        else if (Level2PosAffect.condition(dValue))  return  2;
        else if (Level3PosAffect.condition(dValue))  return  3;
        else if (Level4PosAffect.condition(dValue))  return  4;
        else if (Level5PosAffect.condition(dValue))  return  5;
        else if (Level6PosAffect.condition(dValue))  return  6;
        return 0;   // in dead-band: no level triggered
    }

    public static double GetLevelMean(int iLevel1) throws Exception {
        switch (iLevel1) {
            case -6: return (Level6NegAffect.MAX_VALUE + Level6NegAffect.MIN_VALUE) / 2.0;
            // ... cases -5 through +6 ...
            case  6: return (Level6PosAffect.MAX_VALUE + Level6PosAffect.MIN_VALUE) / 2.0;
        }
        throw new Exception("Invalid level: " + iLevel1);
    }
}
```

5.4 Level1PosAffectPlan — The Cursor Monitor Plan

```
public plan Level1PosAffectPlan extends Plan {
    #handles event SetAffectCursorsEvent ev1;
    #posts  event LevelCursorEvent lce;
    #uses interface NeonAffectAgent enc;

    // Only this plan handles cursor #1
    static boolean relevant(SetAffectCursorsEvent ev1) {
        return ev1.iCursorNumber == 1;
    }

    body() {
```

```
        // Create the Change cursor bound to the live emotion variable
        Level1PosAffect c = new Level1PosAffect(enc.emotionCursorVar, enc.GetAgent());

        while(true) {
            @wait_for(c);                                // sleep until condition() == true
            enc.SetCursorAck(c.GetCursorType(), c.GetLevel(), true);
            @post(lce.post(CursorEnum.iLEVEL_CURSOR_POS_1)); // fire level event
            @wait_for(mutex6.planWait());                // wait for downstream to finish
        }
    }
}
```

There are 12 such plans (Level1PosAffectPlan through Level6PosAffectPlan, and Level1NegAffectPlan through Level6NegAffectPlan), all running concurrently. Multiple cursors can be simultaneously in-range when the emotion value is near a boundary, which is why LevelCursorPlan waits for **AckCursor** — a cursor that fires only when all 12 monitors have reported their acknowledgement.

5.5 LevelCursorPlan — Central Routing Plan

```
public plan LevelCursorPlan extends Plan {
    #handles event LevelCursorEvent lce;
    #posts   event AddAffectHistoryEvent aahe;
    #posts   event CursorPlanChoiceEvent cpce;
    #uses interface NeonAffectAgent enc;

    body() {
        int    iLevel   = lce.iLevel;

        // Read current affect type and emotion value from beliefsets
        affectType.get(0, $affect).next();
        int    iAffect  = $affect.as_int();
        emotion.get(0, $emotion).next();
        double dEmotion = $emotion.as_double();

        // Record this event in the affect history
        @post(aahe.post(1, System.currentTimeMillis(), iAffect, dEmotion));
        @wait_for(mutex3.planWait());

        // Wait until ALL 12 level cursors have acknowledged
        Cursor c2 = new AckCursor(enc.ackCursorVar, enc.GetNumberOfCursors());
        @wait_for(c2);

        // Send to behavior selection
        @post(cpce.post(iLevel, iAffect, dEmotion));
        @wait_for(mutex3.planWait());
        // mutex6.signal() — unblocks all cursor plans
    }
}
```

5.6 CursorPlanChoicePlan — Deciding the Behavior Category

Once all cursor acknowledgements are received, **CursorPlanChoicePlan** consults the **NeonAffectDeciderAgent** via a **DecideEvent/DecideReply** handshake. The decider returns one of three modes:

- **bLevelOnly** — use only the current intensity level for plan selection
- **bLevelAndAffect** — use level + the specific emotion type

• **bLevelAndExtreme** — use level + affect + extreme-threshold logic

Based on the decider's response and the *iLevel* value, CursorPlanChoicePlan posts the appropriate level event — e.g. **Level6NegAffectEvent(iAffect)** for RAGE at level -6 — and calls **@subtask(levelEvent)** to execute the corresponding behavior plan (such as **SetPlanContextRagePlan**).

6. The GetBehavior Method — AbstractMiniAgent

AbstractMiniAgent.java is the base class for all 155 emotion-specific mini-agents. Its **GetBehavior(String strModeratorType)** method orchestrates the entire behavior-selection pipeline and is the central method of the Neon-Affect architecture.

```
public agent AbstractMiniAgent extends Agent {
    #has capability WaitCapability wc;
    #has capability AbstractAffectCapability cap1;
    #has capability GetSetFormulasCapability gsfc;
    #private data PlansWithFormulas plansFormulas();
    #private data PlanContextName   planContextName();
    #private data Behavior          behaviorName();
    #agent  data  Semaphore         mutex();
    #private data Semaphore         mutex2();

    private  int    iAffectType      =  -1;
    private String strInteractionType = "";
    private ArrayList plansWithFormulas = new ArrayList();
    private  ArrayList  selectedPlans   =  new  ArrayList();
    private String    strBehavior       = "";

    public String GetBehavior(String strModeratorType) {
        try {
            // Step 1: enumerate all plans known to this agent
            ArrayList planNames = GetPlans();

            // Step 2: filter to plans that have non-zero PLAN_VALUE in cojack_cognitive.xml
            ArrayList planNamesWithFormulas =
                GetPlansWithFormulas(strModeratorType, planNames);

            // Step 3: call OpenAI to rank the filtered plans by percentage
            ArrayList selectedPlans =
                GetSelectedPlans(strInteractionType, planNamesWithFormulas);

            // Step 4: write the selected/non-selected formula into each plan's
            //         CoJACK cognitive slot
            for (int i = 0; i < selectedPlans.size(); i++) {
                Object[] items       = (Object[])selectedPlans.get(i);
                String strPlanName   = (String)items[0];
                String strPlanFormula = (String)items[1];
                SetCognitiveFormula(strModeratorType, strPlanName, strPlanFormula);
            }

            // Step 5: open CoJACK plan competition (ALL_PLANS context)
            postEvent(spce2.post(PlanConstants.ALL_PLANS));
            mutex.threadWait();

            // Step 6: post the emotion-specific level event (implemented by subclass)
            PostEvent();
```

```
            // Step 7: read the winning behavior name from the Behavior beliefset
            postEvent(gbe.post(iAffectType));
            mutex.threadWait();
        }
        catch (Exception e) {
            e.printStackTrace();
            mutex.signal();
        }
        return strBehavior;
    }
}
```

6.1 Step 1 — GetPlans()

```
private ArrayList GetPlans() {
    ArrayList ret = new ArrayList();
    Plan[] plans = cap1.getAgent().getKnownPlans();
    for (int i = 0; i < plans.length; i++) {
        Plan   plan1       = plans[i];
        String strPlanName = plan1.getPlanName();
        if (PlanUtils.ContainsCorrectPlan(strPlanName)) {
            ret.add(strPlanName);
        }
    }
    return ret;
}
```

This iterates the JACK agent's known-plans list and filters it through **PlanUtils.ContainsCorrectPlan()**, which checks that the plan name matches the naming convention expected for this emotion group (e.g. plans whose names contain "Rage" for a NeonRageAgent).

6.2 Step 2 — GetPlansWithFormulas()

```
private ArrayList GetPlansWithFormulas(String strModeratorType, ArrayList planNames) {
    // Fire event to GetSetFormulasCapability → GetFormulasPlan
    postEvent(gsfc.gfc.gfe.post(strModeratorType, planNames));
    mutex.threadWait();

    // Retrieve the filtered list from the PlansWithFormulas beliefset
    postEvent(gpfe.post(iAffectType));
    mutex.threadWait();

    return plansWithFormulas;
}
```

6.3 Step 3 — GetSelectedPlans()

```
private ArrayList GetSelectedPlans(String strInteractionType, ArrayList planNames) {
    postEvent(gspe.post(iAffectType, strInteractionType, planNames));
    mutex.threadWait();
    return selectedPlans;
}
```

7. Reading cojack_cognitive.xml — GetFormulasPlan

Before calling OpenAI, the system must determine which plans are *eligible* for selection — i.e. which plans have been pre-configured in **cojack_cognitive.xml** with a non-zero **PLAN_VALUE**. This is the role of **GetFormulasPlan**, which uses the **CoJACKConfigParserXML** library to parse the XML at runtime:

```
public plan GetFormulasPlan extends Plan {
    #handles event GetFormulasEvent gfe;
    #uses data PlansWithFormulas plansFormulas;

    body() {
        String    strModeratorType = gfe.strModeratorType;  // e.g. "RageNonverbal"
        ArrayList planNames        = gfe.planNames;

        for (int i = 0; i < planNames.size(); i++) {
            String strPlanName          = (String)planNames.get(i);
            // Construct the parameter group name: "Neon" + moderatorType
            String strParameterGroupName = "Neon" + strModeratorType;
            // Strip namespace prefix for XML lookup
            String strConfigPlanName     =
                strPlanName.replace("NeonAffect.", "");

            String strPlanNameValue = "0";   // default: not configured

            try {
                // Parse cojack_cognitive.xml and look up the plan's PLAN_VALUE
                strPlanNameValue = new parser(Constants.COJACK_COGNITIVE_PATH)
                    .GetPlanNameValue(strParameterGroupName, strConfigPlanName);
            }
            catch (Exception e) { e.printStackTrace(); }

            if (!strPlanNameValue.equals("0")) {
                // This plan has a non-zero PLAN_VALUE — include it as a candidate
                plansFormulas.add(plansFormulas.nFacts(), strPlanName);
            }
        }
    }
}
```

The lookup path is: **cojack_cognitive.xml → parameter_group[name="NeonRageNonverbal"] → param[name="ScreamPlan"]/@value**. Only plans with a value other than "0" are considered valid candidates. This allows the system designer to enable or disable specific behaviors per emotion and interaction type simply by editing the XML — no recompilation required.

7.1 cojack_cognitive.xml — Full Parameter Structure

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration start_time="0.0">
  <parameter_group name="default">

    <!-- ■■ Global CoJACK cognitive parameters ■■ -->
    <param name="ACTIVATION_NOISE"            value="0.001"/>
    <param name="PLAN_VALUE"                  value="0"/>
    <param name="PARTIAL_MATCHING"            value="false"/>
    <param name="PLAN_SUCCESS_PROBABILITY"    value="1.0"/>
    <param name="K_DEPTH"                     value="2.0"/>
    <param name="MAXIMUM_ASSOCIATIVE_STRENGTH" value="0.0"/>
    <param name="SIMILAR_BELIEFS_MEMORY"      value="2.0"/>
    <param name="MISMATCH_SIMILARITY"         value="0.0"/>
```

```
    <param name="DECAY_RATE"                    value="0.75"/>
    <param name="PLAN_ACTIVATION_NOISE"         value="0.001"/>
    <param name="EVENT_SIGNIFICANCE"            value="20"/>
    <param name="LATENCY_EXPONENT"              value="1.0"/>
    <param name="MISMATCH_PENALTY"              value="1.0"/>
    <param name="GOAL_ACTIVATION"               value="1.0"/>
    <param name="EVENT_SOURCE_SCALING_FACTOR" value="0.0"/>
    <param name="PLAN_SUCCESS_COST"             value="10"/>
    <param name="DEFAULT_ACTION_TIME"           value="0.050"/>
    <param name="EXPECTED_GAIN_NOISE"           value="1.0"/>
    <param name="BASE_LEVEL_ACTIVATION"         value="0.5"/>
    <param name="PLAN_PERMANENT_ACTIVATION_NOISE" value="0.001"/>
    <param name="BASE_LEVEL_CONSTANT"           value="0.01"/>
    <param name="LATENCY_FACTOR"                value="0.05"/>
    <param name="ACTIVATION_THRESHOLD"          value="0.0"/>
    <param name="MAXIMUM_DIFFERENCE"            value="-1.0"/>
    <param name="PERMANENT_ACTIVATION_NOISE"   value="0.001"/>

    <!-- ■■ Master emotional reservoir ■■ -->
    <parameter_group name="NeonAffectGroup">
      <param name="mbti_personality_type" value="INTP"/>
      <param name="neon_parameter"          value="1"/>
      <moderator events_decay_time="0.0"
        reservoir_formula="if ((reservoir - 0.2525 * dt > 0) &amp;amp;&amp;amp; reservoir > 0.05)
          then (reservoir - 0.2525 * dt)
          else if ((reservoir + 0.2525 * dt &lt; 0) &amp;amp;&amp;amp; reservoir &lt; -0.05)
          then (reservoir + 0.2525 * dt) else 0.0;"
        type="NeonAffectReservoir">
        <param name="reservoir" value="0"/>
        <applies_to formula="reservoir" name="neon_parameter"/>
      </moderator>
    </parameter_group>

    <!-- ■■ Per-emotion, per-modality parameter groups ■■ -->
    <!-- (One group for each of the 155 emotions × 3 modalities = 465 groups) -->

    <parameter_group name="NeonRageVerbal">
      <param name="rage_verbal_parameter" value="1"/>
      <moderator events_decay_time="0.0"
          reservoir_formula="if (reservoir - 0.2525 * dt > 0) then
              (reservoir - 0.2525 * dt) else 0.0;"
          type="Value">
        <param name="reservoir" value="0"/>
        <applies_to formula="reservoir" name="rage_verbal_parameter"/>
      </moderator>
      <!-- Each eligible plan has a PLAN_VALUE entry: -->
      <!-- <param name="ScreamPlan"      value="0.8"/> -->
      <!-- <param name="CursePlan"       value="0.6"/> -->
    </parameter_group>

    <parameter_group name="NeonRageNonverbal">...</parameter_group>
    <parameter_group name="NeonRageAction">...</parameter_group>
    <!-- ... repeated for all 155 emotions ... -->

  </parameter_group>
</configuration>
```

7.2 CoJACK Cognitive Parameter Explanations

ACTIVATION_NOISE (0.001): Random Gaussian noise added to each plan's activation score at every evaluation cycle. Provides stochasticity so identical situations don't always produce identical behavior.

PLAN_VALUE (0): Base utility value for all plans. Individual plans override this in their parameter group. The expected gain formula is: E[G] = P(success) × PLAN_VALUE − PLAN_SUCCESS_COST.

DECAY_RATE (0.75): ACT-R-style base-level learning decay exponent. Controls how quickly rarely-used plans lose activation over time.

EVENT_SIGNIFICANCE (20): Weight multiplier for events in the plan activation calculation. Higher values make recent events more influential in plan selection.

BASE_LEVEL_ACTIVATION (0.5): Starting activation for all plans before any history-based adjustment. Acts as a prior belief in plan usability.

EXPECTED_GAIN_NOISE (1.0): Scale of noise added to the expected gain computation, preventing the same plan from always winning ties.

PLAN_SUCCESS_PROBABILITY (1.0): Prior probability that any plan will succeed when executed. Used in: E[G] = this × PLAN_VALUE − cost.

PLAN_SUCCESS_COST (10): Fixed cost subtracted from expected gain. Plans with PLAN_VALUE < 10 have negative expected gain and will never be selected.

DEFAULT_ACTION_TIME (0.050): Default execution time in seconds for primitive actions within plans, used in latency calculations.

LATENCY_FACTOR (0.05) + LATENCY_EXPONENT (1.0): Parameters for the ACT-R latency equation: T = F × e^(-A) where A is activation and F is the factor.

ACTIVATION_THRESHOLD (0.0): Plans with activation below this threshold are excluded from consideration. Set to 0.0 here so all plans compete.

K_DEPTH (2.0): Depth of the spreading activation network. Controls how far associative activation propagates through related beliefs.

8. OpenAI API Integration

Two separate OpenAI LLM calls are made per behavior-selection cycle. Both calls are personality-aware: they receive the agent's **MBTI type** (stored in cojack_cognitive.xml as mbti_personality_type = INTP) and the **AffectHistory** — a timestamped log of past emotions and behaviors — to enable context-sensitive responses.

8.1 First OpenAI Call — DetermineAffectContext (InteractionPlan)

This call occurs inside **SetPlanContextRagePlan** (and its equivalents for each emotion), which sends a **ContextEvent** to **NeonContextComputeAgent**. **InteractionPlan** handles the event:

```
public plan InteractionPlan extends Plan {
    #handles event ContextEvent ce;
    #sends  event ContextReply  reply;
    #uses interface NeonInteractionComputeAgent enc;

    body() {
        boolean bVerbal    = true;   // default
        boolean bNonverbal = false;
```

```
        boolean bAction    = false;

        @action() {
            Object[] items = enc.GetNeonAffectAgent().GetAffectHistoryBehaviors();
            String strPersonalityType = enc.GetNeonAffectAgent().GetPersonalityType();

            ArrayList affectHistory = (ArrayList)items[0];
            ArrayList behaviorList  = (ArrayList)items[1];

            if (affectHistory.size() != 0 && behaviorList.size() != 0) {
                String strPrompt = CreateOpenAIPrompt(
                    affectHistory, behaviorList, strPersonalityType);

                // ■■ FIRST OPENAI CALL ■■
                ArrayList ret = new openai().DetermineAffectContext(strPrompt);

                for (int i = 0; i < ret.size(); i++) {
                    String strResult = (String)ret.get(i);
                    if      (strResult.equals("Verbal"))    bVerbal    = true;
                    else if (strResult.equals("Nonverbal")) bNonverbal = true;
                    else if (strResult.equals("Action"))    bAction    = true;
                }
            }
        };

        @reply(ce, reply.response(bVerbal, bNonverbal, bAction));
    }

    private String CreateOpenAIPrompt(ArrayList affectHistory, ArrayList behaviorList,
                                      String strPersonalityType) {
        String strPrompt =
            "Given this history of emotions and behaviors, determine whether to do a " +
            "\"verbal,\" \"nonverbal,\" or \"action\" given an " +
            strPersonalityType + " Myers-Briggs personality type. " +
            "Give only one answer for the entire list.\n";

        long now = System.currentTimeMillis();
        for (int i = 0; i < behaviorList.size(); i++) {
            Object[] items       = (Object[])behaviorList.get(i);
            String   strBehavior = (String)items[0];
            long     time        = (long)items[1];
            double   dDuration   = (now - time) / 1000.0;

            AffectHistoryVar ahv = (AffectHistoryVar)affectHistory.get(0);
            int iType = ahv.GetType();

            strPrompt += (i+1) + ".  A " + AffectEnum.GetName(iType) +
                " emotion occurred with a " + strBehavior +
                " behavior at " + dDuration + " seconds ago.\n";
        }
        return strPrompt;
    }
}
```

Example prompt sent to OpenAI (first call):

```
Given this history of emotions and behaviors, determine whether to do a "verbal,"
"nonverbal," or "action" given an INTP Myers-Briggs personality type.
Give only one answer for the entire list.
```

```
1.  A rage emotion occurred with a ScreamPlan behavior at 45.0 seconds ago.
2.  A rage emotion occurred with a ClenchFistsPlan behavior at 120.0 seconds ago.
```

Expected response: Nonverbal

The response directly determines which *interaction type* mini-agent is created: a **NeonRageAgent** is instantiated with strInteractionType = "Nonverbal". Each of the three possible responses leads to a different parameter group being consulted in cojack_cognitive.xml (NeonRageVerbal, NeonRageNonverbal, or NeonRageAction).

8.2 Second OpenAI Call — DetermineSelectedPlans (PlanSelectorPlan)

```
public plan PlanSelectorPlan extends Plan {
    #handles event PlanSelectorEvent pse;
    #sends   event PlanSelectorReply reply;
    #uses interface NeonPlanSelectorAgent enc;

    body() {
        ret2 = new ArrayList();
        ThreadPool tp = enc.GetThreadPool();

        // Execute in thread pool (non-blocking for JACK scheduler)
        @action(tp) {
            String strEmotion         = pse.strEmotion;          // e.g. "rage"
            String strInteractionType = pse.strInteractionType;  // e.g. "nonverbal"
            ArrayList planNames       = pse.planNames;

            Object[] items = enc.GetNeonAffectAgent().GetAffectHistoryBehaviors();
            String strPersonalityType = enc.GetNeonAffectAgent().GetPersonalityType();
            ArrayList affectHistory   = (ArrayList)items[0];
            ArrayList behaviorList    = (ArrayList)items[1];

            String strPrompt = CreateOpenAIPrompt(
                planNames, affectHistory, behaviorList,
                strEmotion, strInteractionType, strPersonalityType);

            // ■■ SECOND OPENAI CALL ■■
            ArrayList ret = new openai().DetermineSelectedPlans(strPrompt);

            // Find the highest percentage returned
            int iMaxPercentage = 0;
            for (int i = 0; i < ret.size(); i++) {
                Object[] p2     = (Object[])ret.get(i);
                String   strPct = ((String)p2[1]).replace("%", "");
                int      iPct   = Integer.parseInt(strPct);
                if (iPct > iMaxPercentage) iMaxPercentage = iPct;
            }

            // Tag each plan as SELECTED or NON-SELECTED
            for (int i = 0; i < ret.size(); i++) {
                Object[] p2      = (Object[])ret.get(i);
                String   pName   = ((String)planNames.get(i))
                                      .replace(Constants.NEON_NAMESPACE, "").trim();
                String   strPct  = ((String)p2[1]).replace("%", "");
                int      iPct    = Integer.parseInt(strPct);

                Object[] out = new Object[2];
                out[0] = pName;
                out[1] = (iPct == iMaxPercentage)
```

```
                        ? PlanConstants.RANDOM_SELECTED_PLAN      // winner
                        : PlanConstants.RANDOM_NON_SELECTED_PLAN; // loser
                ret2.add(out);
            }
        };
        @reply(pse, reply.response(ret2));
    }

    private String CreateOpenAIPrompt(ArrayList planNames, ArrayList affectHistory,
        ArrayList behaviorList, String strEmotion,
        String strInteractionType, String strPersonalityType) {

        strEmotion         = strEmotion.toLowerCase();
        strInteractionType = strInteractionType.toLowerCase();

        String strPrompt =
            "Given these \"" + strEmotion + "\", " + strInteractionType +
            " behaviors, provide the usage percentages as a numbered list with behavior " +
            "name and usages, of what an " + strPersonalityType +
            " Myers-Briggs personality type would do:\n";

        for (int i = 0; i < planNames.size(); i++) {
            String n = ((String)planNames.get(i))
                        .replace(Constants.NEON_NAMESPACE, "").trim()
                        .replace(strEmotion, "").trim();
            n = PlanUtils.InsertSpaces(n);
            strPrompt += (i+1) + ".  " + n + "\n";
        }
        strPrompt += "Do not add to this list.\n" +
                     "Please respond like this: 1. Crying Pain - 15%\n";
        return strPrompt;
    }
}
```

Example prompt sent to OpenAI (second call):

```
Given these "rage", nonverbal behaviors, provide the usage percentages as a numbered
list with behavior name and usages, of what an INTP Myers-Briggs personality type
would do:
1.  Scream
2.  Clench Fists
3.  Stomp
4.  Break Object
Do not add to this list.
Please respond like this: 1. Crying Pain - 15%
```

Example OpenAI response:

```
1. Scream - 10%
2. Clench Fists - 35%
3. Stomp - 15%
4. Break Object - 40%
```

The highest-percentage plan (**Break Object — 40%**) receives the **RANDOM_SELECTED_PLAN** formula tag. All others receive **RANDOM_NON_SELECTED_PLAN**. If two plans share the maximum percentage, both receive RANDOM_SELECTED_PLAN and CoJACK's stochastic selection (EXPECTED_GAIN_NOISE) breaks the tie.

9. Setting CoJACK Cognitive Formulas

After OpenAI returns the ranked plan list, **GetBehavior()** iterates the results and calls **SetCognitiveFormula()** for each plan. This dynamically updates the plan's **PLAN_VALUE** inside the CoJACK cognitive engine, biasing the expected-gain competition toward the OpenAI-selected plan.

```
// In AbstractMiniAgent.GetBehavior():
for (int i = 0; i < selectedPlans.size(); i++) {
    Object[] items        = (Object[])selectedPlans.get(i);
    String strPlanName    = (String)items[0];
    String strPlanFormula = (String)items[1]; // RANDOM_SELECTED_PLAN or RANDOM_NON_SELECTED_PLAN
    SetCognitiveFormula(strModeratorType, strPlanName, strPlanFormula);
}

// SetCognitiveFormula routes through the JACK event system:
public void SetCognitiveFormula(String strModeratorType, String strPlanName,
                                String strFormula) {
    postEvent(gsfc.sfc.scfe.post(strModeratorType, strPlanName, strFormula));
    mutex.threadWait();    // block until SetCognitiveFormulaPlan completes
}
```

9.1 RANDOM_SELECTED_PLAN vs. RANDOM_NON_SELECTED_PLAN

These two constants define the CoJACK formula strings that are injected into the cognitive engine:

- **RANDOM_SELECTED_PLAN**: Sets the plan's PLAN_VALUE to a high random number (e.g. in the range [50, 100]). With PLAN_SUCCESS_PROBABILITY = 1.0 and PLAN_SUCCESS_COST = 10, a PLAN_VALUE of 75 gives E[G] = 75 - 10 = 65, which easily dominates all other plans.
- **RANDOM_NON_SELECTED_PLAN**: Sets the plan's PLAN_VALUE to 0. This gives E[G] = 0 - 10 = -10, effectively disqualifying the plan from the CoJACK competition.

The use of a *random* high value (rather than a fixed constant) provides natural variation: when two plans share the maximum OpenAI percentage, their PLAN_VALUEs are independently sampled, and CoJACK's stochastic selection mechanism chooses between them in a probabilistically meaningful way.

9.2 Opening the Plan Competition

```
// Step 5 in GetBehavior() — set plan context to ALL_PLANS
postEvent(spce2.post(PlanConstants.ALL_PLANS));
mutex.threadWait();
```

Setting the **PlanContextName** beliefset to **ALL_PLANS** is the signal that all candidate plans should compete. Each behavior plan's **context()** clause checks this beliefset:

```
// ScreamPlan.java — context clause
context() {
    planContextName.get(0, $name) &&
    ($name.as_string().equals(PlanConstants.ALL_PLANS) ||
     $name.as_string().equals(getPlanName()));
}
```

A plan is eligible to be selected only when its context clause succeeds. By setting PlanContextName to ALL_PLANS, all plans in the group become simultaneously eligible, and CoJACK's expected-gain mechanism selects among them based on the PLAN_VALUEs we just injected.

10. CoJACK Behavior Selection — The Final Competition

10.1 PostEvent() — Firing the Level Event

```
// NeonRageAgent.java — overrides AbstractMiniAgent.PostEvent()
public void PostEvent() {
    try {
        // Post Level6NegAffectEvent for RAGE
        postEvent(ev1.post(GetAffectType()));
        mutex.threadWait();
    }
    catch (Exception e) { e.printStackTrace(); }
}
```

Each emotion agent overrides **PostEvent()** to post the level event appropriate for its emotion type. For RAGE at level -6, this is a **Level6NegAffectEvent**. This event triggers all plans that handle it and have passing context() clauses.

10.2 ScreamPlan — A Behavior Plan

```
public plan ScreamPlan extends Plan {
    #handles event Level6NegAffectEvent ev14;
    #uses data PlanContextName planContextName;
    #uses data Behavior        behaviorName;
    #uses data Semaphore       mutex;

    // Only responds to RAGE
    static boolean relevant(Level6NegAffectEvent ev14) {
        return ev14.iAffect == AffectEnum.RAGE;
    }

    // Context: plan competition is open (ALL_PLANS) OR this plan is specifically named
    context() {
        planContextName.get(0, $name) &&
        ($name.as_string().equals(PlanConstants.ALL_PLANS) ||
         $name.as_string().equals(getPlanName()));
    }

    body() {
        System.out.println("Inside Scream plan.");
        behaviorName.add(0, getPlanName());   // record winning behavior name
        mutex.signal();                        // release AbstractMiniAgent.GetBehavior()
    }

    logical String $name;
}
```

10.3 GetBehaviorPlan — Reading the Result

```
public plan GetBehaviorPlan extends Plan {
    #handles event GetBehaviorEvent gbe;
    #uses interface AbstractMiniAgent enc;
    #uses data Behavior  behaviorName;
    #uses data Semaphore mutex;

    body() {
```

```
        logical String $behaviorName;
        behaviorName.get(0, $behaviorName).next();
        String strBehavior = $behaviorName.as_string();
        enc.SetBehavior(strBehavior);    // write result back to AbstractMiniAgent
        mutex.signal();
    }
}
```

After the winning behavior plan's body() writes its name to the **Behavior** beliefset and signals the mutex, **GetBehaviorPlan** reads that name and calls **enc.SetBehavior()**, which stores it in the **strBehavior** field of the AbstractMiniAgent. The final return strBehavior in **GetBehavior()** then delivers this string back to **SetPlanContextRagePlan**, which adds it to the NeonAffectAgent's behavior history via **enc.AddBehavior()**.

10.4 CoJACK Expected-Gain Decision Logic

The CoJACK engine selects the plan with the highest expected gain, computed as:

```
E[Gain] = PLAN_SUCCESS_PROBABILITY * PLAN_VALUE - PLAN_SUCCESS_COST
          + ACTIVATION_NOISE * gaussian_noise()
          + EXPECTED_GAIN_NOISE * gaussian_noise()

With default parameters and RANDOM_SELECTED_PLAN (e.g. PLAN_VALUE = 75):
  E[G_selected]     = 1.0 * 75 - 10 + noise ≈  65 + small noise
  E[G_non_selected] = 1.0 *  0 - 10 + noise ≈ -10 + small noise

The selected plan wins with overwhelming probability.
```

11. End-to-End Trace — RAGE at Reservoir -60.0

This section walks through every step of the complete behavior-selection pipeline for a RAGE stimulus that drives the reservoir to -60.0 (Level -6).

Step 1 *[Program.java]* — neonAgent.AddEmotionDirect(AffectEnum.RAGE) posts AddEmotionEvent with strModerator="NeonAffectReservoir", dAmount=-60.0, bIncremental=false.

Step 2 *[AddEmotionPlan]* — Configuration.moderatorCurrentReservoir() reads current reservoir (0.0). Because bIncremental=false, sets dReservoir=-60.0. Configuration.moderatorReservoirAgentUpdate() stores -60.0 at current time.

Step 3 *[cojack_cognitive.xml formula]* — CoJACK begins applying the reservoir formula every dt seconds: since reservoir=-60.0 < -0.05 and reservoir+0.2525*dt < 0, the formula decays it upward at 0.2525 units/sec.

Step 4 *[GrabReservoirValuePlan]* — Every 500ms, enc.GetReservoirValue() reads the decayed reservoir value (e.g. -59.87 after 500ms) and calls enc.SetEmotionValue(-59.87), updating EmotionCursorVar.

Step 5 *[Level6NegAffect cursor]* — Level6NegAffectPlan's @wait_for(c) wakes because -65.0 < -59.87 <= -55.0 satisfies condition(). enc.SetCursorAck(LEVEL_CURSOR_NEG_6, -6, true) is called.

Step 6 *[LevelCursorEvent(-6)]* — Level6NegAffectPlan posts LevelCursorEvent(iLEVEL_CURSOR_NEG_6). LevelCursorPlan wakes, reads iAffect=RAGE, dEmotion=-59.87. Posts AddAffectHistoryEvent.

Step 7 *[AckCursor]* — LevelCursorPlan waits for AckCursor — all 12 level cursor plans acknowledge their current state (only Level6Neg is true; others are false). AckCursor fires.

Step 8 *[CursorPlanChoiceEvent(-6, RAGE, -59.87)]* — LevelCursorPlan posts CursorPlanChoiceEvent. CursorPlanChoicePlan sends DecideEvent to NeonAffectDeciderAgent → receives DecideReply(bLevelAndAffect=true).

Step 9 *[Level6NegAffectEvent(RAGE)]* — CursorPlanChoicePlan posts Level6NegAffectEvent(RAGE) and calls @subtask — this triggers SetPlanContextRagePlan.

Step 10 *[ContextEvent → InteractionPlan]* — SetPlanContextRagePlan sends ContextEvent to NeonContextComputeAgent. InteractionPlan builds history prompt and calls new openai().DetermineAffectContext(prompt).

Step 11 *[OpenAI Call #1]* — OpenAI receives the history prompt for an INTP personality. Returns: "Nonverbal". bNonverbal=true.

Step 12 *[NeonRageAgent creation]* — SetPlanContextRagePlan creates NeonRageAgent("NeonRageAgentNonverbal", "Nonverbal", debug) and calls agent.GetBehavior("RageNonverbal").

Step 13 *[GetPlans()]* — GetPlans() returns all known plans for NeonRageAgent: ["NeonAffect.ScreamPlan", "NeonAffect.ClenchFistsPlan", "NeonAffect.StompPlan", "NeonAffect.BreakObjectPlan"].

Step 14 *[GetFormulasPlan]* — parser(cojack_cognitive.xml).GetPlanNameValue("NeonRageNonverbal", "ScreamPlan") returns non-zero for all four plans. All four are added to plansWithFormulas.

Step 15 *[GetSelectedPlansEvent → PlanSelectorPlan]* — GetSelectedPlansPlan forwards to NeonPlanSelectorAgent. PlanSelectorPlan builds the percentage prompt and calls new openai().DetermineSelectedPlans(prompt).

Step 16 *[OpenAI Call #2]* — OpenAI returns: 1. Scream - 10%, 2. Clench Fists - 35%, 3. Stomp - 15%, 4. Break Object - 40%. Maximum is 40%.

Step 17 *[Formula tagging]* — BreakObjectPlan → RANDOM_SELECTED_PLAN. ScreamPlan, ClenchFistsPlan, StompPlan → RANDOM_NON_SELECTED_PLAN.

Step 18 *[SetCognitiveFormula × 4]* — GetBehavior() calls SetCognitiveFormula for each plan. CoJACK updates PLAN_VALUE: BreakObjectPlan gets ~75; others get 0.

Step 19 *[SetPlanContextEvent2(ALL_PLANS)]* — PlanContextName beliefset updated to ALL_PLANS. All four plans now have passing context() clauses.

Step 20 *[PostEvent() → Level6NegAffectEvent(RAGE)]* — NeonRageAgent.PostEvent() posts Level6NegAffectEvent(RAGE). All four plans are triggered. CoJACK computes E[G] for each.

Step 21 *[CoJACK selects BreakObjectPlan]* — E[G_BreakObject] ≈ 65+noise vs E[G_others] ≈ -10+noise. BreakObjectPlan wins. body() writes "BreakObjectPlan" to Behavior beliefset, signals mutex.

Step 22 *[GetBehaviorPlan]* — Reads Behavior beliefset → "BreakObjectPlan". Calls enc.SetBehavior("BreakObjectPlan"). mutex.signal().

Step 23 *[Return]* — GetBehavior() returns "BreakObjectPlan". SetPlanContextRagePlan calls enc.AddBehavior("BreakObjectPlan") to update history.

Step 24 *[Reservoir continues decaying]* — Meanwhile, CoJACK continues applying the NeonAffectReservoir moderator formula. Reservoir decays from -60.0 toward 0 at 0.2525/sec. After ~237 seconds it reaches the dead-band and snaps to 0.

12. MBTI Personality Types and Emotion Level Mappings

The MBTI personality type is stored as a CoJACK parameter and read by **NeonAffectAgent.GetPersonalityType()**. It is passed to both OpenAI calls, allowing the LLM to tailor behavior selection to the agent's psychological profile.

```
<!-- In cojack_cognitive.xml NeonAffectGroup: -->
<param name="mbti_personality_type" value="INTP"/>
```

12.1 OrigAffectLevels — 20 Timelines

OrigAffectLevels.java maps each combination of (timeline_id, level) to an **AffectEnum** constant. There are 20 timelines, each representing a different emotional dimension. Examples:

```
// Timeline 1 — Anger/Awe axis
{ 1, -6, AffectEnum.RAGE         }
{ 1, -5, AffectEnum.GRIEF        }
{ 1, -4, AffectEnum.DISTRESS     }
{ 1, -3, AffectEnum.DISCOMFORT   }
{ 1, -2, AffectEnum.BOREDOM      }
{ 1, -1, AffectEnum.MILD_UNEASE  }
{ 1,  1, AffectEnum.INTEREST     }
{ 1,  2, AffectEnum.CONTENTMENT  }
{ 1,  3, AffectEnum.PLEASURE     }
{ 1,  4, AffectEnum.JOY          }
{ 1,  5, AffectEnum.SERENITY     }
{ 1,  6, AffectEnum.AWE          }

// Timeline 2 — Fear/Eagerness axis
{ 2, -6, AffectEnum.OVERWHELMED }
{ 2, -5, AffectEnum.DREAD       }
...
{ 2,  5, AffectEnum.EAGERNESS    }
```

12.2 INTP_AffectLevels — Personality-Specific Overrides

```
// INTP_AffectLevels.java — personality type uses timeline ID = -1
{ -1, -6, AffectEnum.OBSESSED     }
{ -1, -6, AffectEnum.OVERWHELMED }
{ -1, -6, AffectEnum.POWERLESS    }
{ -1,  6, AffectEnum.AWE            }
{ -1,  6, AffectEnum.CONNECTEDNESS  }
{ -1,  6, AffectEnum.ENJOYMENT      }
{ -1,  6, AffectEnum.EUPHORIA       }
{ -1,  6, AffectEnum.MASTERFUL      }
{ -1,  6, AffectEnum.TRIUMPHANT     }
{ -1,  6, AffectEnum.WONDER         }
```

For INTPs, extreme negative states include OBSESSED and POWERLESS in addition to OVERWHELMED, reflecting the INTP tendency toward rumination under stress. Extreme positive states include MASTERFUL and WONDER, reflecting INTP cognitive strengths.

12.3 AffectHistory BeliefSet

AffectHistory.java stores a timestamped, leveled log of all emotional events and serves as the context window for both OpenAI calls:

```
// AffectHistory beliefset schema:
//   int  iID     — sequential identifier
//   int  iLevel  — level (-6 to +6) at which this event occurred
//   int  iAffect — AffectEnum constant (e.g. RAGE = 3)
//   double dValue — reservoir value at the time of the event
//   long   time   — System.currentTimeMillis() when recorded
```

13. Key BeliefSets, Events, and Plans Summary

13.1 BeliefSets

BeliefSet	Fields	Purpose
Emotion	int id, double dAmount	Current emotional intensity (reservo: r snapshot)
AffectType	int id, int iAffect	Current affect type (AffectEnum cons ant)
Levels	int id, int level	Current discrete level (-6 to +6)
Behavior	int id, String strName	Selected behavior plan name
PlanContextName	int id, String strName	Active plan context (ALL_PLANS or sp cific name)
PlansWithFormulas	int id, String strPlanName	Plans eligible for OpenAI ranking
AffectHistory	int iID, int iLevel, int iAffect, d	Tuibmlestdeaplaeude,emlotnigontailmeevent log
Appraisals	24 float fields: pleasantness, thre	OtC,C-mreordtealinatpyp,raeitsca.l dimensions (futur e use)

13.2 Key Events

Event	Fields	Triggered By
AddEmotionEvent	String strModerator, double dAmount	NeboncAlffbeIontcArgeemnetn.tAadldEmotionDirect)
GrabReservoirValueEvent	(BDIGoalEvent)	NeonAffectAgent.PostInit()
LevelCursorEvent	int iLevel	Level{N}[Pos|Neg]AffectPlan
CursorPlanChoiceEvent	int iLevel, int iAffect, double dEm	LteivoenlCursorPlan
Level6NegAffectEvent	int iAffect	CursorPlanChoicePlan
ContextEvent	(empty)	SetPlanContextRagePlan
ContextReply	bool bVerbal, bNonverbal, bAction	InteractionPlan
GetFormulasEvent	String strModeratorType, ArrayList	pGleatnPNlaamnesWithFormulas()
GetSelectedPlansEvent	int iAffect, String strInteractionT	Gpet,SeAlrercatyeIdiPsltanpsl(a)ns
PlanSelectorEvent	String strEmotion, String strIntera	GteitcSneTlyepcet,edAPrlranysLPilsatnplanNames
PlanSelectorReply	ArrayList selectedPlans	PlanSelectorPlan
GetBehaviorEvent	int iAffect	AbstractMiniAgent.GetBehavior()
SetPlanContextEvent2	String strName	AbstractMiniAgent.GetBehavior()
DecideEvent	(empty)	CursorPlanChoicePlan
DecideReply	bool bLevelOnly, bLevelAndAffect, b	NeevoenlAfnfdeEcxtDrecmiederAgent

14. Conclusions

Neon-Affect represents a sophisticated integration of three distinct AI paradigms: **BDI agent reasoning** (JACK), **cognitive science-grounded decision making** (CoJACK / ACT-R), and **large language model intelligence** (OpenAI GPT). The architecture is notable for several design decisions:

- **Separation of concerns**: The CoJACK reservoir handles continuous emotional dynamics; the level cursor system handles discretisation; OpenAI handles contextual plan ranking; CoJACK's expected-gain mechanism handles the final stochastic selection. No single component carries more responsibility than it should.
- **Personality as a first-class parameter**: The MBTI type is stored in the cognitive XML and flows through both OpenAI prompts, ensuring that behavioral selection is personality-consistent across the full pipeline.
- **Dynamic formula injection**: Rather than bypassing CoJACK's plan-selection mechanism, the system cooperates with it by injecting OpenAI's recommendations as PLAN_VALUE updates. CoJACK still makes the final decision, preserving cognitive plausibility and enabling the stochastic variation that BDI models require.
- **Dead-band reservoir decay**: The 0.2525 units/second decay with ±0.05 dead-band is a careful engineering choice that prevents emotional values from lingering near zero indefinitely while also preventing oscillation around the neutral point.
- **Scale**: 155 emotions × 3 modalities × up to ~20 behaviors each = 9,300 potential plan configurations, all governed by a single XML file and two OpenAI prompts per cycle.

This paper was generated from direct source code analysis of the tenbase2-com-LLC/Neon-Affect GitHub repository. All code excerpts are taken directly from the repository source files.