FOREX EA GENERATION SYSTEM

Closed-Loop Autonomous EA Development Platform

MASTER DOCUMENT • Vision + Build Specification • v2.0

Confidential — Internal Use Only

Document Version	2.0 — Incorporates all design revisions
Status	ACTIVE — Primary build reference
Section 1	Vision & Business Context (human + Claude)
Section 2	Build Specification (Claude build instructions)
Author	Lou Lucarelli, powered by Claude

How To Use This Document

Section 1 — The Vision: Read this first. It explains what we are building, why it matters, and where it is going. This section is appropriate to share with partners and potential investors. It gives Claude the business context needed to make good decisions during development.

Section 2 — The Build Specification: This is the Lego instructions. It is written for Claude. At the start of every development session, provide Claude with this full document and state which phase you are working on. Claude will have complete context with no gaps to fill.

SECTION 1

The Vision

What we are building, why it matters, and where it is going

1. The Problem We Are Solving

Building a Forex Expert Advisor (EA) today is a chore. Not because the work is intellectually hard — it is not. It is a chore because of the mechanics. Write a prompt, wait for code, copy it into MetaEditor, compile it, watch for errors, copy those errors back, get a fix, compile again, run a backtest, wait, transcribe eleven numbers into a chat window, discuss, get a new version, repeat. A single EA takes four to eight weeks of calendar time, hundreds of iterations, and enormous human patience.

The bottleneck is not the AI. The AI can generate code in seconds. The bottleneck is the human acting as a messenger between the AI and the tools — copy/pasting code, transcribing results, reporting back. This is clerical work that produces no insight and consumes the time that should be spent on thinking.

This system eliminates the clerical loop entirely. It connects the AI directly to MetaTrader 5, lets it write code, compile it, run backtests, read the results, and decide what to try next — all without a human touching a keyboard. The human re-enters the process only when genuine judgment is required: when something unexpected happens, when a strategic direction needs to be set, or when the system has done everything it can autonomously and needs a different perspective.

2. What We Are Building

The system has three components that work together as a team.

The MT5 Bridge

A Python program that talks directly to MetaTrader 5. It can deploy an EA file, trigger a compilation, run a backtest over any time period, and extract the results — all programmatically. This is the component that eliminates copy/paste. A full iteration cycle that currently takes thirty to sixty minutes of human time runs in two to three minutes automatically.

The Orchestrator

The project manager. A Python program that runs the development loop for each EA. It knows what EA is being built, what has been tried, what the results were, and what should happen next. It routes tasks to the right AI model, logs everything to a structured registry, and decides when to keep going autonomously versus when to escalate to the human. It does not have opinions — it executes a decision framework.

The LLM Think Tank

Three AI models, each used for what it does best. GPT-4o writes clean, compilable MQ5 code. Claude reasons about strategy architecture, diagnoses why an EA is stuck, and writes the escalation briefs. Gemini holds the full iteration history in its large context window and identifies patterns across dozens of runs. None of them talk to each other directly — the Orchestrator is the intermediary for every exchange.

3. How a Single Iteration Works

Here is one complete loop, in plain English, from start to finish.

Step	Who Acts	What Happens
1	Orchestrator	Wakes up. Checks the EA Registry. Knows it is on iteration 14 of Mean Reversion EURUSD 4H. Best result so far: 7.4% annual return, 13.8% drawdown.
2	Claude	Orchestrator asks: given the last 5 results, what should we change? Claude reads the metrics and recommends adding a volatility regime filter using ATR.
3	GPT-4o	Orchestrator asks: here is the current code, add an ATR-based volatility filter. GPT-4o writes the updated MQ5 file.
4	MT5 Bridge	Bridge checks the code for look-ahead bias. Compiles it. If compile errors exist, they are sent back to GPT-4o automatically and fixed without human involvement.
5	MT5 Bridge	Bridge triggers a backtest: EURUSD, 4H, 8 years. MetaTrader runs it and produces results.
6	Orchestrator	Results are parsed into structured JSON and logged to the EA Registry. Iteration 15: ATR filter added, return 8.1%, drawdown 11.2%.
7	Claude	Orchestrator asks: should we keep going? Claude sees clear improvement and says continue.
8	Orchestrator	Loop repeats from Step 2.

4. When the Human Gets Involved

The system is designed to work autonomously the vast majority of the time. The human is called in for a specific, narrow set of situations — not to rubber-stamp every decision, but to provide judgment that the system genuinely cannot produce on its own.

Trigger	What It Means	What You Do
Benchmark floor not reached	SMA Cross on GBPJPY returning 3% after 18 iterations. Expected floor is 8%. This is a logic problem, not a parameter problem.	Open a Claude conversation pre-loaded with full context. Diagnose together. Agree on a direction.
Walk-forward degradation	EA performed well in-sample but collapsed on the hidden 2-year test period. Likely overfit.	Decide: simplify the strategy, adjust parameters, or retire this approach.
Suspicious metrics	Win rate jumped to 79% in 3 iterations. May indicate curve-fitting.	Review with Claude. Decide whether to run additional validation or revert.

Genuine plateau	No meaningful improvement after 20+ iterations despite logic and parameter changes.	Strategic direction change needed. Only a human can decide what that direction is.
Milestone reached	EA passed all validation gates and is ready for the live portfolio.	Review and approve graduation.

When escalation happens, you receive a Slack message with a link. That link opens a Claude conversation that is already pre-loaded with the full history of the EA, the reason for escalation, and a diagnosis. You do not start from zero. You ask questions, push back, think out loud, and arrive at a clear direction. Claude summarizes that direction as a precise instruction that goes back to the Orchestrator. The loop resumes.

5. The Three AI Models and Their Roles

A note for Claude reading this document: the model routing described below is intentional. You are not the right tool for every task, and neither is GPT-4o. Use this section to understand your specific role in the system and why the boundaries are drawn where they are.

Model	Role	Why This Model
GPT-4o	MQ5 code generation and compile error fixes	Best syntax fidelity for MQL5. Produces clean, compilable code consistently. Does not hallucinate MT5 function signatures as frequently as other models.
Claude	Strategic reasoning, iteration decisions, escalation briefs, overfitting diagnosis	Stronger at understanding why a strategy is failing, not just what the metrics say. Better at architectural reasoning and knowing when to escalate versus continue.
Gemini 1.5 Pro	Iteration memory and pattern recognition	1 million token context window. Can hold 50+ backtest results and identify trends that would be invisible in a shorter context. Free API tier is sufficient for this role.

6. Safeguards Built Into the System

Automation without safeguards produces fast wrong answers. The system has several layers of protection that run automatically, without requiring human involvement.

Data Quality Pre-Flight

Every backtest starts with a data integrity check. Before a single iteration runs, the bridge validates that the historical data for the target symbol and timeframe is complete. Gaps in tick data — especially during high-volatility periods like news events — are a silent, invisible failure mode. An EA backtested on data with a four-hour gap never gets tested against the hardest market conditions. It can look excellent in testing and fail immediately live. This check is a hard stop: if the data does not pass, iteration does not begin.

Look-Ahead Bias Detection

Every MQ5 file is scanned before it reaches MetaTrader. Look-ahead bias occurs when an EA's signal logic references price data from bars that have not yet closed — data that would not have been available at the time of the trade in real markets. MetaTrader will run a backtest on a look-ahead biased EA without complaint, producing results that cannot be replicated live. The bridge scans every file for these patterns and blocks any file that fails from reaching the backtester.

Cross-Pair Pulse Check

Starting at iteration 15, a lightweight cross-pair check runs every 10 iterations. A strategy that genuinely captures an edge should show some signal on a correlated pair — not identical results, but a recognizable pattern. An EA that produces 8% on GBPJPY but 0.4% on USDJPY at iteration 20 is a strong overfitting signal. Catching this at iteration 20 saves potentially 80 more iterations of wasted optimization. A full cross-pair validation suite runs again at graduation.

Walk-Forward Validation

The last two years of data are locked away from the optimizer from the beginning. The EA never sees this data during development. When it reaches the graduation gate, it is run against this hidden period for the first time. If its performance degrades by more than 40% compared to the in-sample period, it does not graduate. This is the single most important check in the system — it is the difference between a beautiful backtest and a strategy that works in live markets.

7. The EA Portfolio and Business Goal

No single EA achieves the target in isolation. The goal is a collection of uncorrelated strategies that together produce a smooth, compounding equity curve. Each EA does one thing well, on one or two pairs, at one timeframe. The portfolio effect — running strategies that do not move together — is what produces the target performance characteristics.

Target	Value	How Achieved
Annual Return	40%	Portfolio of 6-8 uncorrelated EAs, each contributing 6-12% individually
Max Drawdown	<15%	Correlation monitoring ensures EAs do not all draw down simultaneously
Equity Curve	Smooth	Strategy diversity across types, pairs, and timeframes reduces stagnation
Sharpe Ratio	>1.5	Low correlation between EAs raises portfolio Sharpe above any individual EA

EA Development Roadmap

EA	Type	Primary Pair	Status
SMA Cross	Trend Following	GBPJPY H1	✅ Graduated — 8% annual return
Mean Reversion	Mean Reversion	EURUSD H4	🔄 In Development

Breakout / Vol Expansion	Breakout	GBPUSD H1	📋 Queued
Session Bias	Time-Based	USDJPY H1	📋 Queued
News Event Strangle	Event-Driven	Multi-pair	📋 Queued
Regime-Switching	Adaptive	EURUSD H4	📋 Queued

8. The Long-Term Business: PAMM/MAM

The EA portfolio is the foundation of a PAMM/MAM managed account business targeting high-income earners and international investors seeking proven, systematic Forex diversification. The business model is straightforward: 20% performance fee on net new profits, high water mark, zero management fee. Investors only pay when they make money.

The system being built here is not just an EA development tool — it is the R&D infrastructure for that fund. The EA Registry becomes the trade attribution system. The walk-forward results become the performance disclosure material. The escalation logs become the risk management documentation. Every architectural decision made during development should be made with this end state in mind.

Why This System Enables the Business

Traditional EA development is too slow to build and maintain a diversified, live portfolio. By reducing development time from 4-8 weeks to 3-7 days per EA, the system makes it practical to build, validate, retire, and replace EAs continuously — treating the portfolio as a living system rather than a fixed product. That is what institutional-quality systematic trading requires, and it is what this platform is designed to produce.

SECTION 2

The Build Specification

Lego instructions for Claude. Read Section 1 first for business context.

Instructions for Claude

This section is your build guide. Before writing any code, read Section 1 in full.
It contains the business context that explains why decisions are made the way they are.

This document defines the complete system. Work through it phase by phase.
Each phase ends with a COMPLETION CHECKPOINT — a specific, runnable test that confirms the phase is done before moving to the next one. Do not skip checkpoints.

When a design decision is not covered here, refer to Section 1 for intent, then ask the founder before proceeding. Do not assume. Do not fill gaps silently.

The founder is not a Python expert. Write code that is readable, well-commented, and structured so a non-expert can understand what each file does and why.

9. System Architecture

9.1 Repository Structure

Every file has a single, clear responsibility. Do not combine responsibilities into one file. When in doubt, create a new file.

forex_ea_system/ — Full Repository Layout

```
forex_ea_system/
├── main.py                    # Entry point. Run this to start the system.
│
├── orchestrator/
│   ├── __init__.py
│   ├── session.py             # Manages one EA development session end-to-end
│   ├── decision.py            # The decision state machine — continue/pivot/escalate
│   ├── registry.py            # Read and write EA Registry JSON files
│   └── portfolio.py           # Correlation monitor across graduated EAs
│
├── mt5_bridge/
│   ├── __init__.py
│   ├── connection.py          # MT5 initialize/shutdown with retry logic
│   ├── data_validator.py      # Pre-flight data quality check — hard stop
│   ├── compiler.py            # MetaEditor compile + error capture + auto-retry
│   ├── lookahead_guard.py     # Scan MQ5 code for look-ahead bias before compile
│   ├── backtest.py            # Trigger backtest and return structured JSON
│   └── schemas.py             # BacktestResult dataclass definition
│
├── llm/
│   ├── __init__.py
│   ├── router.py              # Routes tasks to the correct model
│   ├── anthropic_client.py    # Claude API wrapper
│   ├── openai_client.py       # GPT-4o API wrapper
│   └── gemini_client.py       # Gemini 1.5 Pro API wrapper
│
├── prompts/
│   ├── mq5_generator.json     # GPT-4o: generate MQ5 code
│   ├── compile_fix.json       # GPT-4o: fix compile errors
│   ├── lookahead_fix.json     # GPT-4o: fix look-ahead bias violations
│   ├── iteration_decision.json# Claude: should we continue, pivot, or escalate?
│   ├── architecture_review.json# Claude: diagnose a stuck EA
│   └── escalation_brief.json  # Claude: write a human escalation brief
│
├── validation/
│   ├── __init__.py
│   ├── walk_forward.py        # Walk-forward validation runner
│   ├── monte_carlo.py         # Monte Carlo simulation on trade sequence
│   └── cross_pair.py          # Cross-pair pulse check and full validation
│
├── mq5_library/
│   ├── core/
│   │   ├── order_management.mqh
│   │   ├── position_sizing.mqh
│   │   ├── session_filter.mqh
│   │   └── trade_logging.mqh
│   ├── indicators/
│   │   ├── sma_cross.mqh      # Includes 3-candle lookback as default option
│   │   ├── bollinger_bands.mqh
│   │   ├── atr.mqh
│   │   └── rsi.mqh
│   ├── filters/
│   │   ├── volatility_regime.mqh
```

```
│   │   ├── spread_filter.mqh
│   │   └── news_filter.mqh
│   └── templates/
│       ├── trend_following.mq5
│       ├── mean_reversion.mq5
│       └── breakout.mq5
│
├── escalation/
│   ├── __init__.py
│   ├── notifier.py            # Send Slack or email alert
│   └── brief_builder.py       # Assemble escalation brief with full context
│
├── registry/
│   └── eas/                   # One JSON file per EA
│       └── sma_cross_gbpjpy_1h.json   # Example: existing graduated EA
│
└── config/
    ├── settings.py            # API keys, MT5 path, all thresholds
    ├── model_config.json      # Task-to-model routing table
    ├── domain_benchmarks.json # Strategy/pair minimum performance floors
    └── graduation_criteria.json
```

9.2 Data Flow

Understanding how data moves through the system prevents integration errors. Every component has one input type and one output type.

Data Flow Through the System

```
INPUT: EA specification (strategy type, pair, timeframe, params)
   ↓
data_validator.py   →  DataQualityReport  →  PASS or HARD STOP
   ↓ (pass only)
llm/router.py       →  GPT-4o             →  MQ5 code (string)
   ↓
lookahead_guard.py  →  GuardReport        →  PASS or send back to GPT-4o
   ↓ (pass only)
compiler.py         →  CompileResult      →  PASS or send errors to GPT-4o
   ↓ (pass only)
backtest.py         →  BacktestResult     →  JSON metrics
   ↓
registry.py         →  EA Registry        →  Updated JSON file
   ↓
decision.py         →  Action             →  CONTINUE / PIVOT / ESCALATE / GRADUATE
   ↓
session.py          →  Next iteration OR escalation brief
```

10. Phase 1 — The MT5 Bridge

Phase Goal

By the end of Phase 1, Python can: connect to MT5, validate data quality, deploy an EA, compile it, scan for look-ahead bias, run a backtest, and return a structured JSON result. Every step that was previously done by hand is now done by code. Human copy/paste is completely eliminated.

10.1 MT5 Connection

Build this first. Prove the connection works before building anything on top of it.

mt5_bridge/connection.py

```
# mt5_bridge/connection.py
import MetaTrader5 as mt5
import time
import logging

logger = logging.getLogger(__name__)

def initialize(max_retries: int = 3, retry_delay: float = 2.0) -> bool:
    """
    Connect to a running MT5 terminal.
    MT5 must already be open and logged into a broker account.
    Returns True on success, raises RuntimeError on failure.
    """
    for attempt in range(1, max_retries + 1):
        if mt5.initialize():
            info = mt5.terminal_info()
            logger.info(f'MT5 connected: {info.name} build {info.build}')
            return True
        error = mt5.last_error()
        logger.warning(f'MT5 init attempt {attempt}/{max_retries} failed: {error}')
        if attempt < max_retries:
            time.sleep(retry_delay)
    raise RuntimeError(
        f'Could not connect to MT5 after {max_retries} attempts. '
        f'Ensure MT5 is running and logged into your broker account. '
        f'Last error: {mt5.last_error()}'
    )

def shutdown():
    mt5.shutdown()
    logger.info('MT5 connection closed')
```

Verification — Run This Before Continuing

python -c "from mt5_bridge.connection import initialize, shutdown; initialize(); shutdown()"

Expected output: 'MT5 connected: [your broker terminal name] build [number]'
If this fails: MT5 is not running, not logged in, or the MetaTrader5 Python package is not installed. Run: pip install MetaTrader5

10.2 Data Quality Pre-Flight

This is a hard stop. It runs before every new EA session and before any iteration on a new symbol or timeframe. If it fails, the session does not start. Do not add logic to work around a failure — fix the data.

mt5_bridge/data_validator.py

```
# mt5_bridge/data_validator.py
import MetaTrader5 as mt5
import pandas as pd
from dataclasses import dataclass, field
from datetime import datetime
from typing import List

@dataclass
class DataQualityReport:
    symbol:              str
    timeframe:           str
    period_start:        datetime
    period_end:          datetime
    expected_bars:       int
    actual_bars:         int
    completeness_pct:    float
    gaps:                List[dict] = field(default_factory=list)
    largest_gap_mins:    int = 0
    hv_gaps:             List[dict] = field(default_factory=list)
    warnings:            List[str]  = field(default_factory=list)
    verdict:             str = 'PASS'     # 'PASS' or 'FAIL'
    blocking_reason:     str = ''

# Minimum standards per timeframe
THRESHOLDS = {
    'M1':  {'max_gap_mins': 5,   'min_completeness': 99.5},
    'M5':  {'max_gap_mins': 15,  'min_completeness': 99.0},
    'M15': {'max_gap_mins': 30,  'min_completeness': 98.5},
    'H1':  {'max_gap_mins': 120, 'min_completeness': 98.0},
    'H4':  {'max_gap_mins': 480, 'min_completeness': 97.0},
}

def validate(symbol: str, timeframe: str,
             date_from: datetime, date_to: datetime) -> DataQualityReport:
    """
    Check historical data completeness before any backtest session begins.
    HARD STOP: if report.verdict == 'FAIL', do not proceed with iteration.
    """
    tf_constant = getattr(mt5, f'TIMEFRAME_{timeframe}', None)
    if tf_constant is None:
        raise ValueError(f'Unknown timeframe: {timeframe}')

    rates = mt5.copy_rates_range(symbol, tf_constant, date_from, date_to)

    if rates is None or len(rates) == 0:
        return DataQualityReport(
            symbol=symbol, timeframe=timeframe,
            period_start=date_from, period_end=date_to,
            expected_bars=0, actual_bars=0, completeness_pct=0,
            verdict='FAIL',
            blocking_reason=(
                f'No data returned for {symbol} {timeframe}. '
                f'Open MT5, go to Tools > History Center, and download '
                f'data for this symbol before retrying.'
            )
        )
```

```
    df = pd.DataFrame(rates)
    df['time'] = pd.to_datetime(df['time'], unit='s')
    df = df.sort_values('time').reset_index(drop=True)

    tf_mins   = _timeframe_minutes(timeframe)
    expected  = _expected_bars(date_from, date_to, tf_mins)
    actual    = len(df)
    complete  = round((actual / expected) * 100, 2) if expected > 0 else 0

    # Detect gaps (tolerance: 50% over expected bar interval)
    diffs    = df['time'].diff().dt.total_seconds().div(60)
    gap_mask = diffs > (tf_mins * 1.5)
    gaps = []
    for idx in df[gap_mask].index:
        gaps.append({
            'start':    df.loc[idx-1, 'time'].isoformat(),
            'end':      df.loc[idx,   'time'].isoformat(),
            'duration_minutes': int(diffs[idx])
        })

    largest   = max((g['duration_minutes'] for g in gaps), default=0)
    thresh    = THRESHOLDS.get(timeframe, THRESHOLDS['H1'])
    report    = DataQualityReport(
        symbol=symbol, timeframe=timeframe,
        period_start=date_from, period_end=date_to,
        expected_bars=expected, actual_bars=actual,
        completeness_pct=complete, gaps=gaps, largest_gap_mins=largest
    )

    # Apply hard-stop rules
    if complete < thresh['min_completeness']:
        report.verdict = 'FAIL'
        report.blocking_reason = (
            f'Data completeness is {complete}%, below the {thresh["min_completeness"]}% '
            f'minimum for {timeframe}. Re-download historical data in MT5.'
        )
    elif largest > thresh['max_gap_mins']:
        report.verdict = 'FAIL'
        report.blocking_reason = (
            f'Largest data gap is {largest} minutes, exceeding the '
            f'{thresh["max_gap_mins"]} minute maximum for {timeframe}. '
            f'This gap may silently corrupt backtest results.'
        )

    # Advisory warning for suspiciously perfect data
    if complete == 100.0 and len(gaps) == 0 and actual > 10000:
        report.warnings.append(
            'Data shows 100% completeness with zero gaps. '
            'Verify this is genuine tick data, not synthesized history.'
        )
    return report

def _timeframe_minutes(tf: str) -> int:
    return {'M1':1,'M5':5,'M15':15,'M30':30,'H1':60,'H4':240,'D1':1440}[tf]

def _expected_bars(start, end, tf_mins):
    # Approximate: excludes weekends (Forex market closed)
    total_mins = (end - start).total_seconds() / 60
    trading_ratio = 5/7  # 5 trading days out of 7
    return int((total_mins * trading_ratio) / tf_mins)
```

10.3 Look-Ahead Bias Guard

This runs before every compilation. It scans the MQ5 code for indicator calls on shift=0 (the current unclosed bar). If violations are found, the code is sent back to GPT-4o for correction automatically. The backtest never runs on biased code.

mt5_bridge/lookahead_guard.py

```
# mt5_bridge/lookahead_guard.py
import re
from dataclasses import dataclass, field
from typing import List

@dataclass
class GuardReport:
    clean:      bool
    violations: List[dict] = field(default_factory=list)
    verdict:    str = 'PASS'

# Patterns that indicate look-ahead bias in signal generation logic.
# shift=0 means the current bar has not yet closed — its values
# change tick by tick and were not available at signal generation time.
# NOTE: shift=0 is legitimate for reading current spread or open price
# for position sizing. These patterns target signal indicator calls only.
SIGNAL_PATTERNS = [
    (r'iClose\s*\([^)]+,\s*0\s*\)',  'iClose(shift=0) — unclosed bar price'),
    (r'iHigh\s*\([^)]+,\s*0\s*\)',   'iHigh(shift=0)  — unclosed bar high'),
    (r'iLow\s*\([^)]+,\s*0\s*\)',    'iLow(shift=0)   — unclosed bar low'),
    (r'iMA\s*\([^)]+,\s*0\s*\)',     'iMA(shift=0)    — MA on unclosed bar'),
    (r'iRSI\s*\([^)]+,\s*0\s*\)',    'iRSI(shift=0)   — RSI on unclosed bar'),
    (r'iBands\s*\([^)]+,\s*0\s*\)',  'iBands(shift=0) — BB on unclosed bar'),
    (r'iATR\s*\([^)]+,\s*0\s*\)',    'iATR(shift=0)   — ATR on unclosed bar'),
]

def check(mq5_code: str) -> GuardReport:
    """
    Scan MQ5 source code for look-ahead bias patterns.
    Call this before every compilation. If report.clean is False,
    send the violations back to GPT-4o using the lookahead_fix prompt.
    """
    violations = []
    for pattern, description in SIGNAL_PATTERNS:
        matches = re.findall(pattern, mq5_code)
        if matches:
            violations.append({
                'description': description,
                'count':       len(matches)
            })
    clean = len(violations) == 0
    return GuardReport(
        clean=clean,
        violations=violations,
        verdict='PASS' if clean else 'FAIL — LOOK-AHEAD BIAS DETECTED'
    )
```

10.4 Compiler with Auto-Retry

MetaEditor is invoked via subprocess. Compile errors are captured, parsed, and automatically sent back to GPT-4o for correction. This loop runs up to three times before escalating to the human.

mt5_bridge/compiler.py

```
# mt5_bridge/compiler.py
import subprocess, re, logging
from pathlib import Path
from dataclasses import dataclass, field
from typing import List, Optional

logger = logging.getLogger(__name__)

@dataclass
class CompileResult:
    success:  bool
    errors:   List[str] = field(default_factory=list)
    warnings: List[str] = field(default_factory=list)
    raw_log:  str = ''

def compile_ea(mq5_path: Path, metaeditor_path: str) -> CompileResult:
    """Run MetaEditor on the given .mq5 file and return the result."""
    result = subprocess.run(
        [metaeditor_path, '/compile', str(mq5_path), '/log'],
        capture_output=True, text=True, timeout=120
    )
    log      = result.stdout + result.stderr
    errors   = re.findall(r'(?i)error[^\n]*\n', log)
    warnings = re.findall(r'(?i)warning[^\n]*\n', log)
    success  = result.returncode == 0 and len(errors) == 0
    if success:
        logger.info(f'Compiled successfully: {mq5_path.name}')
    else:
        logger.warning(f'Compile failed ({len(errors)} errors): {mq5_path.name}')
    return CompileResult(success=success, errors=errors,
                         warnings=warnings, raw_log=log)

def compile_with_retry(mq5_path: Path, metaeditor_path: str,
                       llm_client, max_retries: int = 3) -> CompileResult:
    """
    Compile with automatic error correction loop.
    If compilation fails, errors are sent to GPT-4o for correction.
    Retries up to max_retries times before raising an exception.
    """
    from mt5_bridge.lookahead_guard import check as guard_check

    for attempt in range(1, max_retries + 1):
        code = mq5_path.read_text(encoding='utf-8')

        # Look-ahead check before every compile attempt
        guard = guard_check(code)
        if not guard.clean:
            logger.info('Look-ahead bias detected — sending to GPT-4o for fix')
            fixed = llm_client.fix_lookahead(code, guard.violations)
            mq5_path.write_text(fixed, encoding='utf-8')
            continue

        result = compile_ea(mq5_path, metaeditor_path)
        if result.success:
            return result

        if attempt < max_retries:
            logger.info(f'Attempt {attempt} failed — sending errors to GPT-4o')
            fixed = llm_client.fix_compile_errors(code, result.errors)
            mq5_path.write_text(fixed, encoding='utf-8')
        else:
```

```
            raise RuntimeError(
                f'Could not compile {mq5_path.name} after {max_retries} attempts. '
                f'Last errors: {result.errors}'
            )
    return result
```

10.5 Backtest Runner and Result Schema

The BacktestResult schema is the lingua franca of the entire system. Every component that produces or consumes metrics uses this exact structure. Do not add fields without updating schemas.py.

mt5_bridge/schemas.py

```
# mt5_bridge/schemas.py
from dataclasses import dataclass, field, asdict
from typing import List
from datetime import datetime
import json

@dataclass
class BacktestResult:
    # Identity
    ea_id:            str
    version:          int
    timestamp:        str
    symbol:           str
    timeframe:        str
    period_start:     str
    period_end:       str

    # Core performance — these are the 11 metrics tracked per iteration
    net_profit_pct:   float
    annual_return:    float
    max_drawdown_pct: float
    sharpe_ratio:     float
    profit_factor:    float
    total_trades:     int
    win_rate_long:    float
    win_rate_short:   float
    avg_trade_pips:   float
    stagnation_days:  int       # Longest drawdown recovery period
    recovery_factor:  float

    # Diagnostic flags (set by guards and validators)
    compile_errors:         List[str] = field(default_factory=list)
    lookahead_violations:   List[str] = field(default_factory=list)
    data_quality_warnings:  List[str] = field(default_factory=list)

    def to_json(self) -> str:
        return json.dumps(asdict(self), indent=2)

    def to_summary(self) -> str:
        """Compact one-line summary for LLM prompts."""
        return (
            f'v{self.version}: return={self.annual_return}% '
            f'dd={self.max_drawdown_pct}% sharpe={self.sharpe_ratio} '
            f'pf={self.profit_factor} trades={self.total_trades} '
            f'win_l={self.win_rate_long}% win_s={self.win_rate_short}%'
        )
```

mt5_bridge/backtest.py

```
# mt5_bridge/backtest.py
import MetaTrader5 as mt5
import logging
from datetime import datetime
from mt5_bridge.schemas import BacktestResult

logger = logging.getLogger(__name__)

def run_backtest(ea_id: str, version: int, ea_name: str,
                 symbol: str, timeframe: str,
                 date_from: datetime, date_to: datetime,
                 params: dict) -> BacktestResult:
    """
    Trigger a Strategy Tester backtest in MT5 and return structured results.
    Assumes mt5_bridge.connection.initialize() has already been called.
    ea_name must match the EA filename in MT5's Experts directory (no .ex5 extension).
    """
    tf_constant = getattr(mt5, f'TIMEFRAME_{timeframe}')

    logger.info(f'Running backtest: {ea_name} {symbol} {timeframe} '
                f'{date_from.date()} to {date_to.date()}')

    result = mt5.strategy_tester_run(
        expert      = ea_name,
        symbol      = symbol,
        period      = tf_constant,
        date_from   = date_from,
        date_to     = date_to,
        optimization= mt5.OPTIMIZATION_DISABLED,
        model       = mt5.MODEL_EVERY_TICK,
        params      = params,
    )

    if result is None:
        raise RuntimeError(
            f'Backtest returned None for {ea_name}. '
            f'MT5 error: {mt5.last_error()}'
        )

    return _parse_result(result, ea_id, version, symbol, timeframe,
                         date_from, date_to)

def _parse_result(raw, ea_id, version, symbol, tf, date_from, date_to):
    """
    Convert MT5's raw tester result into a BacktestResult.
    NOTE: MT5's strategy_tester_run result structure may vary by broker/build.
    Validate this mapping against a known backtest result during Phase 1 testing.
    """
    return BacktestResult(
        ea_id=ea_id, version=version,
        timestamp=datetime.utcnow().isoformat(),
        symbol=symbol, timeframe=tf,
        period_start=date_from.date().isoformat(),
        period_end=date_to.date().isoformat(),
        net_profit_pct = round(raw.get('profit', 0), 2),
        annual_return  = round(_annualize(raw.get('profit',0), date_from, date_to), 2),
        max_drawdown_pct = round(raw.get('max_drawdown', 0), 2),
        sharpe_ratio   = round(raw.get('sharpe_ratio', 0), 2),
        profit_factor  = round(raw.get('profit_factor', 0), 2),
        total_trades   = int(raw.get('trades', 0)),
```

```
        win_rate_long  = round(raw.get('win_rate_long', 0), 1),
        win_rate_short = round(raw.get('win_rate_short', 0), 1),
        avg_trade_pips = round(raw.get('avg_profit_pips', 0), 1),
        stagnation_days= int(raw.get('max_consecutive_loss_days', 0)),
        recovery_factor= round(raw.get('recovery_factor', 0), 2),
    )

def _annualize(total_pct, date_from, date_to):
    years = (date_to - date_from).days / 365.25
    return (total_pct / years) if years > 0 else 0
```

Phase 1 Completion Checkpoint

Run the following against your existing SMA Cross EA.
The numbers returned must match your known manual backtest results.

python -c "
from datetime import datetime
from mt5_bridge.connection import initialize, shutdown
from mt5_bridge.data_validator import validate
from mt5_bridge.backtest import run_backtest
initialize()
report = validate('GBPJPY', 'H1', datetime(2016,1,1), datetime(2024,1,1))
print('Data check:', report.verdict)
result = run_backtest('sma_cross_gbpjpy_1h', 31, 'SMA_Cross',
  'GBPJPY', 'H1', datetime(2016,1,1), datetime(2024,1,1), {})
print(result.to_summary())
shutdown()"

PASS condition: Data check returns PASS. Metrics match your known results.
If metrics do not match: the _parse_result field mapping needs adjustment.
Do not proceed to Phase 2 until this checkpoint passes cleanly.

11. Phase 2 — EA Registry

Phase Goal

Every EA, every version, every backtest result is stored in a structured JSON file.
The Registry is the permanent memory of the system. Chat history is disposable.
The Registry is not.

registry/eas/sma_cross_gbpjpy_1h.json — Populate this manually from your notes

```
// registry/eas/sma_cross_gbpjpy_1h.json — Reference example
{
  "ea_id":           "sma_cross_gbpjpy_1h",
  "strategy_type":   "trend_following",
  "description":     "SMA crossover with 3-candle lookback for intra-candle detection",
  "symbol":          "GBPJPY",
  "timeframe":       "H1",
  "status":          "GRADUATED",
  "created":         "2024-01-01",
  "graduated":       "2024-02-15",
  "iteration_count": 31,
  "current_params":  { "fast_ma": 10, "slow_ma": 50, "lookback_candles": 3 },
  "best_metrics": {
    "annual_return": 8.2, "max_drawdown_pct": 9.1,
    "sharpe_ratio": 1.21, "profit_factor": 1.58, "total_trades": 412
  },
  "iteration_history": [
    { "version": 1,  "change": "Initial SMA cross, no lookback",
      "annual_return": 4.1, "max_drawdown_pct": 14.2, "sharpe_ratio": 0.72 },
    { "version": 12, "change": "Added 3-candle lookback — human escalation insight",
      "annual_return": 6.8, "max_drawdown_pct": 11.3, "sharpe_ratio": 1.04 },
    { "version": 31, "change": "Parameter tuning: fast_ma=10",
      "annual_return": 8.2, "max_drawdown_pct":  9.1, "sharpe_ratio": 1.21 }
  ],
  "escalation_log": [
    { "iteration": 11,
      "reason": "Benchmark floor not reached — 3.1% vs expected 8%",
      "human_decision": "Add 3-candle lookback to capture intra-candle crosses",
      "outcome": "Return improved from 3.1% to 6.8% in next iteration" }
  ],
  "cross_pair_pulses": [
    { "at_iteration": 20, "pair": "USDJPY", "annual_return": 5.1,
      "verdict": "PASS — signal generalizes" }
  ]
}
```

orchestrator/registry.py

```
# orchestrator/registry.py
import json, logging
from pathlib import Path
from datetime import datetime
from mt5_bridge.schemas import BacktestResult

logger    = logging.getLogger(__name__)
REGISTRY  = Path('registry/eas')

def load(ea_id: str) -> dict:
    path = REGISTRY / f'{ea_id}.json'
    if not path.exists():
```

```
        raise FileNotFoundError(f'No registry entry for {ea_id}. Create it first.')
    return json.loads(path.read_text())

def save(ea_state: dict) -> None:
    path = REGISTRY / f'{ea_state["ea_id"]}.json'
    path.write_text(json.dumps(ea_state, indent=2))
    logger.info(f'Registry updated: {ea_state["ea_id"]} v{ea_state["iteration_count"]}')

def log_iteration(ea_id: str, result: BacktestResult, change_description: str) -> None:
    """Append a new iteration result to the EA's history."""
    state = load(ea_id)
    state['iteration_count'] = result.version
    state['iteration_history'].append({
        'version':          result.version,
        'change':           change_description,
        'annual_return':    result.annual_return,
        'max_drawdown_pct': result.max_drawdown_pct,
        'sharpe_ratio':     result.sharpe_ratio,
        'profit_factor':    result.profit_factor,
        'total_trades':     result.total_trades,
    })
    # Update best metrics if this iteration is the best overall
    best = state.get('best_metrics', {})
    if result.annual_return > best.get('annual_return', 0):
        state['best_metrics'] = {
            'annual_return':    result.annual_return,
            'max_drawdown_pct': result.max_drawdown_pct,
            'sharpe_ratio':     result.sharpe_ratio,
            'profit_factor':    result.profit_factor,
            'total_trades':     result.total_trades,
        }
    save(state)

def get_last_n_results(ea_id: str, n: int = 5) -> list:
    """Return the last N iteration summaries for LLM decision prompts."""
    state = load(ea_id)
    return state['iteration_history'][-n:]

def log_escalation(ea_id: str, reason: str, human_decision: str) -> None:
    state = load(ea_id)
    state.setdefault('escalation_log', []).append({
        'iteration':       state['iteration_count'],
        'timestamp':       datetime.utcnow().isoformat(),
        'reason':          reason,
        'human_decision':  human_decision,
    })
    save(state)
```

Phase 2 Completion Checkpoint

1. Manually create registry/eas/sma_cross_gbpjpy_1h.json using the template above. Populate it with the real iteration history from your notes.
2. Run: python -c "from orchestrator.registry import load, get_last_n_results; print(get_last_n_results('sma_cross_gbpjpy_1h', 3))"
3. PASS condition: Returns the last 3 iterations from your JSON file correctly.

12. Phase 3 — LLM Router and Clients

Phase Goal

The Orchestrator can call any of the three AI models by task name.
No component outside the LLM layer knows which model handles which task.
Swapping a model is a one-line config change, not a code change.

config/model_config.json

```
// config/model_config.json
// This file controls which model handles each task.
// To switch a task to a different model, change the value here — not in code.
// Start with full models. Downgrade to mini/haiku only after observing real usage.
{
  "generate_code":        { "provider": "openai",    "model": "gpt-4o" },
  "fix_compile_error":    { "provider": "openai",    "model": "gpt-4o" },
  "fix_lookahead_bias":   { "provider": "openai",    "model": "gpt-4o" },
  "iteration_decision":   { "provider": "anthropic", "model": "claude-sonnet-4-5" },
  "architecture_review":  { "provider": "anthropic", "model": "claude-sonnet-4-5" },
  "write_escalation":     { "provider": "anthropic", "model": "claude-sonnet-4-5" },
  "recall_history":       { "provider": "google",    "model": "gemini-1.5-pro" }
}
```

llm/router.py

```
# llm/router.py
import json
from pathlib import Path
from llm.openai_client   import OpenAIClient
from llm.anthropic_client import AnthropicClient
from llm.gemini_client    import GeminiClient

_config  = json.loads(Path('config/model_config.json').read_text())
_clients = {}   # Lazily initialized — only creates a client when first needed

def _get_client(provider: str):
    if provider not in _clients:
        if provider == 'openai':
            _clients[provider] = OpenAIClient()
        elif provider == 'anthropic':
            _clients[provider] = AnthropicClient()
        elif provider == 'google':
            _clients[provider] = GeminiClient()
        else:
            raise ValueError(f'Unknown provider: {provider}')
    return _clients[provider]

def call(task: str, prompt: str, system: str = '') -> str:
    """
    Route a task to the correct model and return the text response.
    task must match a key in config/model_config.json.
    """
    if task not in _config:
        raise ValueError(f'Unknown task: {task}. Add it to model_config.json.')
    cfg    = _config[task]
    client = _get_client(cfg['provider'])
    return client.complete(prompt=prompt, system=system, model=cfg['model'])
```

llm/anthropic_client.py (and openai_client.py, gemini_client.py)

```
# llm/anthropic_client.py
import anthropic, logging
from config.settings import ANTHROPIC_API_KEY

logger = logging.getLogger(__name__)

class AnthropicClient:
    def __init__(self):
        self.client = anthropic.Anthropic(api_key=ANTHROPIC_API_KEY)

    def complete(self, prompt: str, system: str = '', model: str = 'claude-sonnet-4-5') ->
str:
        logger.info(f'Claude call: {model} ({len(prompt)} chars)')
        message = self.client.messages.create(
            model      = model,
            max_tokens = 4096,
            system     = system,
            messages   = [{ 'role': 'user', 'content': prompt }]
        )
        return message.content[0].text

# llm/openai_client.py  — same pattern
# from openai import OpenAI
# client.chat.completions.create(model=model, messages=[...])

# llm/gemini_client.py  — same pattern
# import google.generativeai as genai
# genai.configure(api_key=GEMINI_API_KEY)
# model = genai.GenerativeModel(model_name)
# model.generate_content(prompt)
```

Phase 3 Completion Checkpoint

python -c "
from llm.router import call
result = call('iteration_decision', 'Respond with exactly: ROUTER_OK')
print(result)"

PASS condition: Prints 'ROUTER_OK'. This confirms Claude API key is valid and routing is working. Repeat with task='generate_code' for OpenAI, and task='recall_history' for Gemini.

13. Phase 4 — The Orchestrator

Phase Goal

The full iteration loop runs automatically. Given an EA ID, the Orchestrator generates code, compiles it, backtests it, logs the result, and decides what to do next — all without human involvement unless an escalation is triggered.

13.1 Decision Logic

This is the most important and most delicate component in the system. It determines when to keep going, when to change direction, and when to call the human. Read Section 1 (specifically sections 4 and 6) before building this.

orchestrator/decision.py

```
# orchestrator/decision.py
import logging
from mt5_bridge.schemas import BacktestResult
from llm import router
from orchestrator import registry

logger = logging.getLogger(__name__)

# ── HARD STOPS ──────────────────────────────────────────────────────
# These are deterministic. If any condition is true, escalate immediately.
# Do not run soft-guidance logic. Do not continue iterating.
HARD_STOPS = [
    (lambda r: r.max_drawdown_pct > 25,
     'Max drawdown exceeded 25%. Structural risk problem — do not iterate further.'),
    (lambda r: r.total_trades < 30,
     'Fewer than 30 trades. Sample size too small for statistical significance.'),
    (lambda r: r.win_rate_long  > 85 or r.win_rate_short > 85,
     'Win rate above 85%. Very likely overfit — results are not credible.'),
]

# ── DOMAIN BENCHMARKS ───────────────────────────────────────────────
# Minimum performance floors by strategy type and symbol.
# If an EA cannot clear its floor after the guard period, something is
# structurally wrong with the logic — not the parameters.
# Add new strategy/pair combinations here as you build the portfolio.
import json
from pathlib import Path
BENCHMARKS = json.loads(Path('config/domain_benchmarks.json').read_text())

def _benchmark_check(ea_state: dict, result: BacktestResult) -> tuple:
    """Returns (triggered: bool, message: str)"""
    strategy = ea_state.get('strategy_type', '')
    symbol   = result.symbol
    bench    = BENCHMARKS.get(strategy, {}).get(symbol)
    if not bench:
        return False, ''
    # Only apply after the guard period (default: 10 iterations)
    guard = bench.get('guard_iterations', 10)
    if ea_state['iteration_count'] < guard:
        return False, ''
    if result.annual_return < bench['min_annual_return']:
        return True, (
```

```
            f'{strategy} on {symbol} returning {result.annual_return}% '
            f'after {ea_state["iteration_count"]} iterations. '
            f'Expected floor: {bench["min_annual_return"]}%. '
            f'This is a logic problem, not a parameter problem.'
        )
    return False, ''

# ── SOFT GUIDANCE ──────────────────────────────────────────────
DECISION_PROMPT = '''
You are reviewing the development progress of a Forex EA.
Based on the last {n} backtest results, recommend exactly ONE action.

Respond with ONLY one of these four words, nothing else:
  CONTINUE       — clear improvement trend exists, keep iterating
  PIVOT_PARAMS   — logic seems sound, but parameters need rethinking
  PIVOT_LOGIC    — parameters are not the problem, the logic needs a shift
  ESCALATE       — you see a pattern you cannot explain from the metrics alone

Rules:
- In the first {guard} iterations, always respond CONTINUE unless a trend is
  clearly negative. Do not trigger pivots during the bootstrapping phase.
- PIVOT_LOGIC means the strategy's fundamental approach needs to change,
  not just the parameters. Only recommend this if parameter tuning is
  demonstrably exhausted.
- ESCALATE means a human needs to see this. Reserve it for genuine anomalies.

EA: {ea_id} | Strategy: {strategy_type} | Pair: {symbol} {timeframe}
Best result so far: {best_metrics}
Last {n} iterations:
{results}
'''

def decide(ea_state: dict, latest_result: BacktestResult) -> tuple:
    """
    Returns (action: str, reason: str)
    action is one of: CONTINUE, PIVOT_PARAMS, PIVOT_LOGIC, ESCALATE, HARD_STOP
    """
    # 1. Hard stops — checked first, always
    for condition, reason in HARD_STOPS:
        if condition(latest_result):
            logger.warning(f'HARD STOP: {reason}')
            return 'HARD_STOP', reason

    # 2. Benchmark floor check
    triggered, message = _benchmark_check(ea_state, latest_result)
    if triggered:
        logger.warning(f'BENCHMARK FLOOR: {message}')
        return 'ESCALATE', message

    # 3. Soft guidance via Claude
    iteration = ea_state['iteration_count']
    guard     = 5   # Do not apply soft rules before iteration 5
    last_5    = registry.get_last_n_results(ea_state['ea_id'], 5)
    results_text = '\n'.join(r.get('change','') + ' → ' +
                             str(r.get('annual_return','?')) + '%'
                             for r in last_5)

    prompt = DECISION_PROMPT.format(
        n=len(last_5), guard=guard,
        ea_id=ea_state['ea_id'],
        strategy_type=ea_state.get('strategy_type',''),
        symbol=ea_state.get('symbol',''), timeframe=ea_state.get('timeframe',''),
        best_metrics=ea_state.get('best_metrics',{}),
```

```
        results=results_text
    )
    action = router.call('iteration_decision', prompt).strip().upper()
    if action not in ('CONTINUE','PIVOT_PARAMS','PIVOT_LOGIC','ESCALATE'):
        logger.warning(f'Unexpected Claude response: {action} — defaulting to CONTINUE')
        action = 'CONTINUE'
    logger.info(f'Decision for {ea_state["ea_id"]}: {action}')
    return action, ''
```

config/domain_benchmarks.json

```
// config/domain_benchmarks.json
{
  "trend_following": {
    "GBPJPY": { "min_annual_return": 8.0,  "guard_iterations": 10 },
    "USDJPY": { "min_annual_return": 6.0,  "guard_iterations": 10 },
    "EURUSD": { "min_annual_return": 5.0,  "guard_iterations": 10 }
  },
  "mean_reversion": {
    "EURUSD": { "min_annual_return": 7.0,  "guard_iterations": 12 },
    "GBPUSD": { "min_annual_return": 6.0,  "guard_iterations": 12 }
  },
  "breakout": {
    "GBPUSD": { "min_annual_return": 8.0,  "guard_iterations": 10 }
  }
}
```

13.2 Session Manager

The session manager runs one complete EA development session — the loop that calls generate, compile, backtest, log, and decide until the session ends.

orchestrator/session.py

```
# orchestrator/session.py
import logging, shutil
from pathlib import Path
from datetime import datetime
from mt5_bridge import connection, data_validator, backtest
from mt5_bridge.compiler import compile_with_retry
from mt5_bridge.schemas import BacktestResult
from orchestrator import registry, decision
from escalation.notifier import escalate
from llm import router
from config.settings import MT5_EXPERTS_DIR, METAEDITOR_PATH
from validation.cross_pair import run_pulse

logger = logging.getLogger(__name__)
CROSS_PAIR_START    = 15   # Start pulse checks at iteration 15
CROSS_PAIR_INTERVAL = 10   # Run every 10 iterations after that

def run_session(ea_id: str, date_from: datetime, date_to: datetime,
                max_iterations: int = 100):
    """
    Run the full development loop for one EA.
    Stops when: max_iterations reached, EA graduates, or unrecoverable error.
    """
    logger.info(f'=== Session started: {ea_id} ===')
```

```
    ea_state   = registry.load(ea_id)
    llm_client = _make_llm_client()   # Wrapper for compile/lookahead fixes

    # ── PRE-FLIGHT ──────────────────────────────────────────────
    connection.initialize()
    dq = data_validator.validate(
        ea_state['symbol'], ea_state['timeframe'], date_from, date_to
    )
    if dq.verdict != 'PASS':
        logger.error(f'DATA PREFLIGHT FAILED: {dq.blocking_reason}')
        escalate(ea_id, 'DATA_QUALITY', dq.blocking_reason)
        connection.shutdown()
        return

    # ── ITERATION LOOP ──────────────────────────────────────────
    for i in range(max_iterations):
        version = ea_state['iteration_count'] + 1
        logger.info(f'--- Iteration {version} ---')

        # Step 1: Ask Claude what to change
        change_spec = _get_next_change(ea_state)

        # Step 2: Ask GPT-4o to write the updated code
        mq5_code = _generate_code(ea_state, change_spec)

        # Step 3: Save code and compile (with auto-retry for errors)
        mq5_path = _save_and_deploy(ea_id, version, mq5_code)
        compile_with_retry(mq5_path, METAEDITOR_PATH, llm_client)

        # Step 4: Run backtest
        result = backtest.run_backtest(
            ea_id=ea_id, version=version,
            ea_name=ea_state['ea_id'],
            symbol=ea_state['symbol'], timeframe=ea_state['timeframe'],
            date_from=date_from, date_to=date_to,
            params=ea_state.get('current_params', {})
        )

        # Step 5: Log result
        registry.log_iteration(ea_id, result, change_spec)
        ea_state = registry.load(ea_id)   # Reload with updated state

        # Step 6: Cross-pair pulse check (from iteration 15, every 10)
        if (version >= CROSS_PAIR_START and
                version % CROSS_PAIR_INTERVAL == 0):
            pulse = run_pulse(ea_state, date_from, date_to)
            if pulse.get('verdict') == 'FAIL':
                escalate(ea_id, 'CROSS_PAIR_FAIL', pulse.get('message'))
                break

        # Step 7: Decide what to do next
        action, reason = decision.decide(ea_state, result)

        if action == 'CONTINUE':
            continue
        elif action in ('HARD_STOP', 'ESCALATE'):
            escalate(ea_id, action, reason)
            break
        elif action in ('PIVOT_PARAMS', 'PIVOT_LOGIC'):
            escalate(ea_id, action,
                f'Claude recommends: {action}. '
                f'Awaiting human direction before continuing.')
            break
```

```
    connection.shutdown()
    logger.info(f'=== Session ended: {ea_id} ===')
```

Phase 4 Completion Checkpoint

Run a 5-iteration session on the Mean Reversion EURUSD EA (create a minimal registry entry first). Watch the log output.

PASS conditions:

1. Data pre-flight runs and logs a result before iteration 1
2. Each iteration logs a version number, a change description, and metrics
3. The EA Registry JSON file is updated after each iteration
4. If you deliberately introduce a compile error in the template, the retry loop catches it and fixes it without stopping the session

Do not proceed to Phase 5 until all four conditions are confirmed.

14. Phase 5 — Escalation System

Phase Goal

When the system needs the human, it sends a structured Slack message with a link to a pre-loaded Claude conversation. The human asks questions, agrees on a direction, and that direction is handed back to the Orchestrator as a precise instruction.

escalation/brief_builder.py

```
# escalation/brief_builder.py
from orchestrator import registry
from llm import router

BRIEF_PROMPT = '''
You are writing an escalation brief for the founder of a Forex EA development system.
The founder is an expert Forex trader, not a Python developer.
Write clearly and concisely. No jargon. No bullet-point walls.

Structure the brief as follows:
  SITUATION  — What is the EA, where is it in development, what happened
  DIAGNOSIS  — Your best assessment of what is causing the issue
  OPTIONS    — Exactly 2 or 3 specific options with trade-offs
               (only if options are appropriate — some escalations are
                informational, not decision points)

End with: 'Reply with your choice, or ask me questions if you need more context.'

EA Registry entry: {registry_json}
Escalation type:   {escalation_type}
Escalation reason: {reason}
'''

def build(ea_id: str, escalation_type: str, reason: str) -> str:
    state = registry.load(ea_id)
    prompt = BRIEF_PROMPT.format(
        registry_json  = str(state),
        escalation_type= escalation_type,
        reason         = reason
    )
    return router.call('write_escalation', prompt)
```

escalation/notifier.py

```
# escalation/notifier.py
import requests, logging
from config.settings import SLACK_WEBHOOK_URL
from escalation.brief_builder import build

logger = logging.getLogger(__name__)

def escalate(ea_id: str, escalation_type: str, reason: str) -> None:
    """
    Build a brief and send it to Slack.
    Also logs to a local file as fallback if Slack is unavailable.
    """
    brief = build(ea_id, escalation_type, reason)
    _log_locally(ea_id, escalation_type, brief)
    if SLACK_WEBHOOK_URL:
```

```
        _send_slack(ea_id, escalation_type, brief)
    else:
        logger.warning('No Slack webhook configured. Brief saved locally only.')
        logger.info(f'ESCALATION BRIEF:\n{brief}')

def _send_slack(ea_id, escalation_type, brief):
    payload = {
        'text': f'*ESCALATION — {ea_id} — {escalation_type}*\n\n{brief}'
    }
    r = requests.post(SLACK_WEBHOOK_URL, json=payload, timeout=10)
    if r.status_code == 200:
        logger.info('Escalation sent to Slack')
    else:
        logger.error(f'Slack send failed: {r.status_code}')

def _log_locally(ea_id, escalation_type, brief):
    from pathlib import Path
    from datetime import datetime
    log_dir = Path('escalations')
    log_dir.mkdir(exist_ok=True)
    ts   = datetime.utcnow().strftime('%Y%m%d_%H%M%S')
    path = log_dir / f'{ts}_{ea_id}_{escalation_type}.txt'
    path.write_text(brief)
```

Phase 5 Completion Checkpoint

Trigger a manual escalation and confirm the brief is useful:

python -c "from escalation.notifier import escalate;
escalate('sma_cross_gbpjpy_1h', 'MILESTONE', 'Manual test escalation')"

PASS conditions:

1. A .txt file is created in the escalations/ directory
2. The brief contains a SITUATION, DIAGNOSIS, and readable content
3. If Slack is configured, the message arrives in your channel
4. The brief reads as if written by someone who knows the EA's history

15. Phase 6 — Overfitting Safeguards

Phase Goal

No EA can graduate without passing walk-forward validation and Monte Carlo. These are hard gates, not advisory checks. An EA that fails either test does not enter the live portfolio regardless of its in-sample metrics.

validation/walk_forward.py

```
# validation/walk_forward.py
from mt5_bridge import backtest, connection
from datetime import datetime

# In-sample: first 75% of history  |  Out-of-sample: last 25% (never seen by optimizer)
# With 8 years of data: IS = 2016-2022, OOS = 2022-2024

DEGRADATION_THRESHOLD = 0.40   # OOS may be up to 40% worse than IS before failing

def run(ea_state: dict, full_start: datetime, full_end: datetime) -> dict:
    total_days = (full_end - full_start).days
    split_date = full_start + __import__('datetime').timedelta(days=int(total_days *
0.75))

    connection.initialize()

    is_result  = backtest.run_backtest(
        ea_id=ea_state['ea_id'], version=0,
        ea_name=ea_state['ea_id'],
        symbol=ea_state['symbol'], timeframe=ea_state['timeframe'],
        date_from=full_start, date_to=split_date,
        params=ea_state['current_params']
    )
    oos_result = backtest.run_backtest(
        ea_id=ea_state['ea_id'], version=0,
        ea_name=ea_state['ea_id'],
        symbol=ea_state['symbol'], timeframe=ea_state['timeframe'],
        date_from=split_date, date_to=full_end,
        params=ea_state['current_params']
    )
    connection.shutdown()

    is_sharpe  = is_result.sharpe_ratio
    oos_sharpe = oos_result.sharpe_ratio
    degradation = (is_sharpe - oos_sharpe) / is_sharpe if is_sharpe > 0 else 1.0

    passed = degradation < DEGRADATION_THRESHOLD
    return {
        'passed':            passed,
        'is_sharpe':         is_sharpe,
        'oos_sharpe':        oos_sharpe,
        'degradation_pct':   round(degradation * 100, 1),
        'verdict': 'ROBUST' if passed else 'OVERFIT — do not graduate'
    }
```

validation/monte_carlo.py

```
# validation/monte_carlo.py
import numpy as np
```

```
N_SIMULATIONS      = 500
MIN_SHARPE         = 0.8
MAX_FAIL_RATE      = 0.30   # More than 30% of sims below MIN_SHARPE = fragile

def run(trades: list) -> dict:
    """
    Shuffle trade sequence 500 times. If Sharpe collapses in >30% of
    simulations, the equity curve depends on trade order — it is fragile.
    trades: list of floats (profit/loss per trade in pips or %)
    """
    base_sharpe = _sharpe(trades)
    fail_count  = sum(
        1 for _ in range(N_SIMULATIONS)
        if _sharpe(np.random.permutation(trades).tolist()) < MIN_SHARPE
    )
    fail_rate = fail_count / N_SIMULATIONS
    passed    = fail_rate < MAX_FAIL_RATE
    return {
        'passed':     passed,
        'base_sharpe':base_sharpe,
        'fail_rate':  round(fail_rate * 100, 1),
        'verdict':    'ROBUST' if passed else 'FRAGILE — do not graduate'
    }

def _sharpe(trades):
    arr = np.array(trades)
    if arr.std() == 0:
        return 0.0
    return float((arr.mean() / arr.std()) * np.sqrt(252))
```

16. Phase 7 — Portfolio Correlation Monitor

Phase Goal

Before any EA is added to the live portfolio, its equity curve is checked against all existing graduated EAs. Maximum allowed pairwise correlation: 0.5. If two EAs are too correlated, they effectively act as one EA in the portfolio.

orchestrator/portfolio.py

```
# orchestrator/portfolio.py
import numpy as np
from pathlib import Path
import json

MAX_CORRELATION = 0.5   # Hard limit on pairwise equity curve correlation

def check_correlation(new_ea_id: str, new_equity_curve: list) -> dict:
    """
    Compare the new EA's equity curve against all graduated EAs.
    Returns a dict with the correlation matrix and a pass/fail verdict.
    Equity curves must cover the same time period for comparison.
    """
    graduated = _load_graduated_curves()
    violations = []

    for ea_id, curve in graduated.items():
        corr = _correlation(new_equity_curve, curve)
        if abs(corr) > MAX_CORRELATION:
            violations.append({
                'ea_id':       ea_id,
                'correlation': round(corr, 2),
                'message': f'{new_ea_id} correlates {corr:.2f} with {ea_id}. '
                           f'Max allowed: {MAX_CORRELATION}.'
            })

    return {
        'passed':     len(violations) == 0,
        'violations': violations,
        'verdict':    'PASS' if not violations else 'FAIL — too correlated'
    }

def _correlation(curve_a, curve_b):
    min_len = min(len(curve_a), len(curve_b))
    a = np.array(curve_a[:min_len])
    b = np.array(curve_b[:min_len])
    if a.std() == 0 or b.std() == 0:
        return 0.0
    return float(np.corrcoef(a, b)[0, 1])

def _load_graduated_curves():
    curves = {}
    for f in Path('registry/eas').glob('*.json'):
        state = json.loads(f.read_text())
        if state.get('status') == 'GRADUATED':
            curve = state.get('equity_curve')
            if curve:
                curves[state['ea_id']] = curve
    return curves
```

17. The MQ5 Snippet Library

The library is one of the highest-value assets in the system. It accumulates every hard-won fix across all EAs in reusable form. GPT-4o is instructed to use library snippets rather than reimplement logic from scratch. This eliminates entire categories of hallucination and keeps EA code consistent.

How GPT-4o Uses the Library

Every code generation prompt includes the list of available snippets.
GPT-4o is instructed: 'Use #include for any library snippet relevant to this strategy. Do not rewrite order management, position sizing, session filtering, or indicator logic that already exists in the library. Write only the strategy-specific logic.'

When a new EA introduces logic that could be reused, Claude flags it for extraction into a new snippet. The library grows with every EA built.

mq5_library/core/order_management.mqh — starter

```
// mq5_library/core/order_management.mqh
// Provides: OpenTrade(), CloseTrade(), ModifyStopLoss()
// All order functions go through this file. Never write order logic inline.
#ifndef ORDER_MANAGEMENT_MQH
#define ORDER_MANAGEMENT_MQH

bool OpenTrade(int orderType, double lotSize, double sl, double tp,
               string comment = "") {
   MqlTradeRequest request = {};
   MqlTradeResult  result  = {};
   request.action    = TRADE_ACTION_DEAL;
   request.type      = (ENUM_ORDER_TYPE)orderType;
   request.volume    = lotSize;
   request.sl        = sl;
   request.tp        = tp;
   request.comment   = comment;
   request.deviation = 10;
   return OrderSend(request, result);
}

#endif
```

mq5_library/indicators/sma_cross.mqh — the intra-candle fix, locked in permanently

```
// mq5_library/indicators/sma_cross.mqh
// Provides: IsCrossUp(), IsCrossDown()
// Uses a configurable lookback window to detect intra-candle crosses.
// Default lookback=3 based on GBPJPY H1 analysis.
// IMPORTANT: Only uses closed bars (shift >= 1). No look-ahead bias.
#ifndef SMA_CROSS_MQH
#define SMA_CROSS_MQH

bool IsCrossUp(int fastPeriod, int slowPeriod, int lookback = 3) {
   for(int i = 1; i <= lookback; i++) {
      double fast_now  = iMA(NULL, 0, fastPeriod, 0, MODE_SMA, PRICE_CLOSE, i);
      double slow_now  = iMA(NULL, 0, slowPeriod, 0, MODE_SMA, PRICE_CLOSE, i);
      double fast_prev = iMA(NULL, 0, fastPeriod, 0, MODE_SMA, PRICE_CLOSE, i+1);
      double slow_prev = iMA(NULL, 0, slowPeriod, 0, MODE_SMA, PRICE_CLOSE, i+1);
      if(fast_prev <= slow_prev && fast_now > slow_now) return true;
```

```
   }
   return false;
}

bool IsCrossDown(int fastPeriod, int slowPeriod, int lookback = 3) {
   for(int i = 1; i <= lookback; i++) {
      double fast_now  = iMA(NULL, 0, fastPeriod, 0, MODE_SMA, PRICE_CLOSE, i);
      double slow_now  = iMA(NULL, 0, slowPeriod, 0, MODE_SMA, PRICE_CLOSE, i);
      double fast_prev = iMA(NULL, 0, fastPeriod, 0, MODE_SMA, PRICE_CLOSE, i+1);
      double slow_prev = iMA(NULL, 0, slowPeriod, 0, MODE_SMA, PRICE_CLOSE, i+1);
      if(fast_prev >= slow_prev && fast_now < slow_now) return true;
   }
   return false;
}

#endif
```

17.1 EA Template Contract System

The system uses bounded mutation zones to ensure EA evolution remains stable, reviewable, and reproducible over thousands of iterations. GPT-4o is not permitted to freely rewrite entire EA architectures. Instead, every EA must conform to a strict template contract with clearly defined mutation regions.

This is a deliberate architectural decision. Unbounded generation produces excessive compile instability, incoherent logic drift, and untraceable mutations over time. The goal of the system is controlled evolutionary improvement, not unrestricted code generation.

Every EA generated by the system must preserve a stable structural framework while allowing controlled mutations inside designated regions only.

Required Mutation Zone Markers

Every EA must contain the following comment markers exactly as written:

C/C++

```
// ENTRY_LOGIC_START
// ENTRY_LOGIC_END

// EXIT_LOGIC_START
// EXIT_LOGIC_END

// FILTER_LOGIC_START
// FILTER_LOGIC_END

// RISK_LOGIC_START
// RISK_LOGIC_END
```

Additional mutation zones may be added later, but these four are the minimum required structure.

Mutation Rules

GPT-4o may only modify code contained inside mutation zones.

The following components are considered protected framework code and may not be modified unless explicitly authorized by the human:

- Order management
- Position sizing framework
- Trade execution functions
- Session management
- Logging infrastructure
- Risk guardrails
- Walk-forward validation hooks
- Cross-pair validation hooks
- Registry integration
- Performance tracking
- Include directives for core library files

The AI must preserve:

- Existing function signatures
- Required include files
- Version headers
- Mutation zone markers
- Compile compatibility with the orchestrator

Mutation Philosophy

The objective is not to generate random strategies from scratch.

The objective is to evolve strategies incrementally while preserving lineage and maintaining measurable cause-and-effect relationships between iterations.

Every iteration should answer a specific research question such as:

- Does adding an ATR volatility filter improve robustness?
- Does extending exit duration reduce drawdown?
- Does RSI confirmation improve trade quality?
- Does session filtering improve Sharpe ratio?

Bounded mutation keeps the evolutionary process understandable and statistically meaningful.

Framework Stability

Each EA version represents a child derived from a known parent version.

Example:

```
C/C++
EA01_V01_002
→
```

```
EA01_V01_003
```

The parent file is never modified in-place. Every mutation creates a brand-new child version with full lineage preserved in:

- run_manifest.csv
- results.csv
- EA Registry history

This guarantees reproducibility, rollback capability, and auditability across the entire development lifecycle.

Validation Enforcement

Before compilation, the orchestrator validates that:

- All required mutation markers exist
- Protected framework regions were not altered
- The EA still conforms to the template contract

If validation fails:

- compilation is blocked
- the iteration is rejected
- the issue is escalated back to GPT-4o for correction

This prevents architecture drift and protects the integrity of the autonomous development system.

18. Configuration and Environment Setup

All sensitive values — API keys, file paths — live in one place. Never hardcode them anywhere else.

config/settings.py

```
# config/settings.py
import os
from pathlib import Path

# ── API KEYS ──────────────────────────────────────────────────────
# Store these in environment variables, not in this file.
# On Windows: set ANTHROPIC_API_KEY=sk-ant-... in System Environment Variables
ANTHROPIC_API_KEY = os.environ.get('ANTHROPIC_API_KEY', '')
OPENAI_API_KEY    = os.environ.get('OPENAI_API_KEY', '')
GEMINI_API_KEY    = os.environ.get('GEMINI_API_KEY', '')
SLACK_WEBHOOK_URL = os.environ.get('SLACK_WEBHOOK_URL', '')

# ── MT5 PATHS ─────────────────────────────────────────────────────
# Adjust these to match your Windows installation
MT5_EXPERTS_DIR = Path(
    r'C:\Users\YOUR_USERNAME\AppData\Roaming\MetaQuotes'
```

```
    r'\Terminal\YOUR_TERMINAL_ID\MQL5\Experts'
)
METAEDITOR_PATH = r'C:\Program Files\MetaTrader 5\metaeditor64.exe'

# ── BACKTEST DEFAULTS ──────────────────────────────────────────
DEFAULT_START = '2016-01-01'
DEFAULT_END   = '2022-01-01'   # Last 2 years reserved for walk-forward OOS
OOS_END       = '2024-01-01'

# ── VALIDATION ON STARTUP ──────────────────────────────────────
def validate():
    missing = [k for k in ['ANTHROPIC_API_KEY','OPENAI_API_KEY','GEMINI_API_KEY']
               if not globals()[k]]
    if missing:
        raise EnvironmentError(
            f'Missing environment variables: {missing}. '
            f'Set them before running the system.'
        )
    if not Path(METAEDITOR_PATH).exists():
        raise FileNotFoundError(
            f'MetaEditor not found at {METAEDITOR_PATH}. '
            f'Update METAEDITOR_PATH in config/settings.py.'
        )
```

19. Quick Reference — EA Lifecycle States

State	Meaning	Next Transition Trigger
INITIALIZING	Registry entry created, first code generation pending	First backtest completes → ITERATING
ITERATING	Active development loop — the system is working autonomously	Metrics pass threshold → VALIDATINGEscalation triggered → ESCALATED
ESCALATED	Awaiting human input — loop is paused	Human provides direction → ITERATINGHuman approves graduation → VALIDATING
VALIDATING	Walk-forward and Monte Carlo running	Both pass → CROSS_TESTINGEither fails → ESCALATED
CROSS_TESTING	Full cross-pair and correlation check	All pass → GRADUATEDFails → ESCALATED
GRADUATED	In the live portfolio	Performance review → stays or RETIRED
RETIRED	Removed from portfolio	Terminal state

20. Quick Reference — Escalation Types

Type	Trigger	System Does	Human Does
HARD_STOP	DD>25%, trades<30, win rate>85%	Pause loop, send brief	Diagnose root cause, decide direction
BENCHMARK_FLOOR	Return below strategy/pair minimum floor	Pause loop, send brief	Review logic with Claude, agree on fix

PLATEAU	Claude returns PIVOT_LOGIC	Pause loop, send brief with options	Choose direction or ask questions
WF_DEGRADATION	Walk-forward fails >40% degradation	Block graduation, send brief	Decide: simplify, adjust, or retire
CROSS_PAIR_FAIL	Pulse check shows no signal on corr pair	Pause loop, send brief	Assess whether overfitting is confirmed
DATA_QUALITY	Pre-flight validator fails	Block session start, send brief	Re-download data in MT5, restart
MILESTONE	Every 10 iters or graduation ready	Send informational brief	Review, no decision required unless graduation

End of Document

Forex EA Generation System • v2.0 • Section 1: Vision • Section 2: Build Specification