LUMINAI.CHAT: Project Technique Document

Version: 2.0 **Last Updated**: 2026-04-20

TABLE OF CONTENTS

EXECUTIVE OVERVIEW

What is luminai.chat?

luminai.chat is NOT just a chat application. It is a **comprehensive AI-powered workspace platform** that integrates:

1. **Conversational AI** - Multi-provider LLM with native tool calling
2. **Code Execution Engine** - Python, JavaScript, MATLAB/Octave in isolated sandboxes
3. **Market Intelligence** - Real-time financial data, quotes, news, economic calendar
4. **Research Platform** - Web search, academic search, learning tools, media processing
5. **Document Management** - Version control, export (DOCX/PDF/TXT), branching history
6. **Scientific Computing** - Full MATLAB/Octave support with advanced graphing
7. **Workplace Features** - File management, team collaboration, credit-based billing

Core Statistics

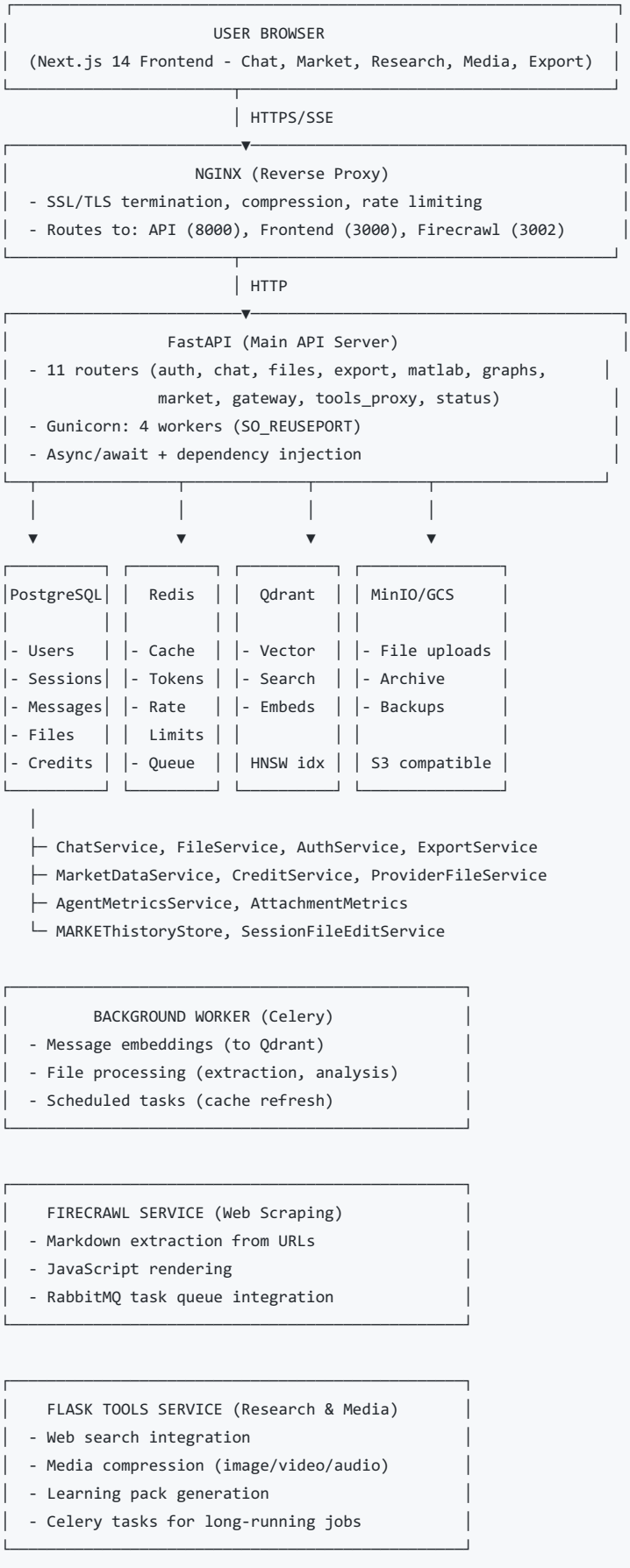
```
Architecture:
├─ 12 Containerized Microservices
├─ 11 API Routers (80+ endpoints)
├─ 15+ Services (business logic)
├─ 12 Database Models (normalized schema)
├─ 18+ Engines & Executors
├─ 50+ Feature Tests + E2E Tests
└─ 35+ Deployment/Automation Scripts

Codebase:
├─ Backend: 20,000+ LOC (Python/FastAPI)
├─ Frontend: 12,000+ LOC (Next.js/TypeScript)
├─ Research Service: 5,000+ LOC (Flask/Celery)
├─ Tests: 8,000+ LOC (pytest/Playwright)
└─ Infrastructure: 2,000+ LOC (Docker/Nginx/Alembic)

Integrations:
├─ LLMs: OpenAI, Anthropic, Google, DeepSeek
├─ Search: Serper API, Perplexity AI, custom Firecrawl
├─ Media: FFmpeg, image processing, audio transcription
├─ Finance: Real-time market data, economic calendar
├─ Web: Firecrawl self-hosted scraping
└─ Security: Cloudflare Turnstile, JWT, Bcrypt
```

COMPLETE ARCHITECTURE (12 MICROSERVICES)

Containerized Stack (docker-compose.yml)

```
# 1. DATABASE LAYER
PostgreSQL 15       # Primary relational database (users, sessions, messages, files, etc.)
Redis 7             # Cache layer + rate limiting + session cache
Qdrant              # Vector database (semantic search, embeddings)
MinIO               # S3-compatible object storage

# 2. API & BACKEND LAYER
FastAPI (api)       # Main API gateway (port 8000, 4 Gunicorn workers)
Worker              # Async background jobs (message embeddings, file processing)

# 3. FEATURES LAYER
MATLAB/Octave       # Scientific computing engine (embedded in API)
Firecrawl           # Web scraping service (port 3002, self-hosted)
RabbitMQ            # Message broker for distributed tasks

# 4. RESEARCH LAYER
Flask (tools_web)   # Secondary web service for research tools (port 5000)
Celery (tools_worker) # Distributed task queue for research jobs

# 5. FRONTEND LAYER
Next.js (web)       # React frontend (port 3000)

# 6. REVERSE PROXY
Nginx               # SSL/TLS termination, routing, rate limiting (port 80/443)
```

Data Flow Architecture



ALL 11 API ROUTERS (80+ ENDPOINTS)

1. Authentication Router (auth.py - 13 endpoints)

```
POST   /api/v1/auth/guest-login          # Anonymous session
POST   /api/v1/auth/login                # Email + password
POST   /api/v1/auth/register             # New account
POST   /api/v1/auth/verify               # Email verification
POST   /api/v1/auth/password-reset       # Forgot password
POST   /api/v1/auth/password-reset/:token # Reset confirmation
POST   /api/v1/auth/refresh              # JWT refresh
GET    /api/v1/auth/me                   # Current user profile
PATCH  /api/v1/auth/me                   # Update profile
PATCH  /api/v1/auth/me/avatar            # Upload avatar
POST   /api/v1/auth/me/change-password   # Change password
POST   /api/v1/auth/me/delete-account    # Account deletion
GET    /api/v1/auth/login-history        # Security audit
```

2. Chat Router (chat.py - 22 endpoints)

```
# Session Management
POST   /api/v1/chat/sessions             # Create session
GET    /api/v1/chat/sessions             # List user sessions
GET    /api/v1/chat/sessions/:id         # Get specific session
PATCH  /api/v1/chat/sessions/:id         # Update session title/settings
DELETE /api/v1/chat/sessions/:id         # Delete session
POST   /api/v1/chat/sessions/:id/branch  # Branch from message

# Message Management
POST   /api/v1/chat/sessions/:id/messages              # Add message + stream agent
GET    /api/v1/chat/sessions/:id/messages              # List messages
GET    /api/v1/chat/sessions/:id/messages/:msg_id      # Get message details
PATCH  /api/v1/chat/sessions/:id/messages/:msg_id      # Edit message
DELETE /api/v1/chat/sessions/:id/messages/:msg_id      # Delete message
POST   /api/v1/chat/sessions/:id/messages/:msg_id/regenerate # Regenerate response

# File Management
POST   /api/v1/chat/sessions/:id/files                 # Attach file to session
GET    /api/v1/chat/sessions/:id/files                 # List session files
DELETE /api/v1/chat/sessions/:id/files/:file_id        # Remove file

# Agent Internals
GET    /api/v1/chat/sessions/:id/agent-logs            # Agent execution logs
GET    /api/v1/chat/sessions/:id/sandbox-contents      # Sandbox file browser
GET    /api/v1/chat/sessions/:id/sandbox/:filename     # Download sandbox file
GET    /api/v1/chat/sessions/:id/tool-results/:result_id # Get tool output
```

3. Files Router (files.py - 2 endpoints)

```
POST   /api/v1/files/upload              # Upload file (multipart/form-data)
GET    /api/v1/files/:file_id/download   # Download file by SHA256 hash
```

4. Export Router (export.py - 3 endpoints)

```
GET    /api/v1/export/formats            # List export formats: [docx, pdf, txt]
POST   /api/v1/export/sessions/:id       # Export session
                                         # Query: ?format=docx|pdf|txt|markdown
GET    /api/v1/export/history            # List previous exports
```

5. Admin Router (admin.py - 7 endpoints)

```
GET    /api/v1/admin/users              # List all users (paginated)
PATCH  /api/v1/admin/users/:id          # Ban/unban, modify credits
POST   /api/v1/admin/users/:id/credits  # Add/subtract credits
POST   /api/v1/admin/system/health      # System health check
GET    /api/v1/admin/logs               # Server logs
GET    /api/v1/admin/stats              # Usage statistics
POST   /api/v1/admin/reset              # System reset (dev only)
```

6. MATLAB Router (matlab.py - 2 endpoints)

```
POST   /api/v1/matlab/execute           # Execute MATLAB/Octave code
GET    /api/v1/matlab/status            # Engine status + available toolboxes
```

7. Graphs Router (graphs.py - 2 endpoints)

```
POST   /api/v1/graphs/execute           # Generate graph from data
GET    /api/v1/graphs/metrics           # Available metrics/chart types
```

8. Market Router (market.py - 10 endpoints)

```
GET    /api/v1/market/symbols           # Search symbols (stocks, crypto)
GET    /api/v1/market/quotes/:symbol    # Real-time price quote
GET    /api/v1/market/ohlc/:symbol      # OHLC historical data (1m to 1Y)
GET    /api/v1/market/news/:symbol      # News for specific symbol
GET    /api/v1/market/flash-news        # Latest market flash news
GET    /api/v1/market/economic-calendar # Economic events + dates
POST   /api/v1/market/search-news       # Full text news search
GET    /api/v1/market/watchlist         # User's watched symbols
POST   /api/v1/market/watchlist         # Add symbol to watchlist
DELETE /api/v1/market/watchlist/:symbol # Remove from watchlist
```

9. LLM Gateway Router (gateway.py - 4 endpoints)

```
GET    /api/v1/gateway/models           # Available models + status
POST   /api/v1/gateway/preference       # Set user's model preference order
GET    /api/v1/gateway/analytics        # Provider usage analytics
POST   /api/v1/gateway/language         # Set UI language preference
```

10. Research Tools Router (tools_proxy.py - 4 endpoints)

```
POST   /api/v1/research/jobs            # Start research job (async)
GET    /api/v1/research/jobs/:id        # Get job status/results
GET    /api/v1/research/jobs            # List user's research jobs
POST   /api/v1/research/format-results  # Format results as learning pack
```

11. Status Router (status.py - 1 endpoint)

```
GET    /api/v1/status                   # Public system status (no auth)
```

ALL 15+ SERVICES

Core Services

1. ChatService

- `create_session()` - Create/branch sessions

- `add_message()` - Store messages with versioning
- `edit_message()` - Update + cascade regeneration
- `branch_session()` - Create alternative branch
- `get_session_history()` - All messages in order
- `delete_session()` - Cleanup all related data

2. AuthService

- `register()` - New account with email verification
- `login()` - JWT token generation
- `verify_email()` - Confirm email address
- `refresh_token()` - Issue new JWT
- `reset_password()` - Password reset flow
- `verify_token()` - JWT validation

3. FileService

- `upload()` - Store file + calculate SHA256 hash
- `deduplicate()` - Check if file exists by hash
- `process_file()` - Extract text/images from document
- `download()` - Retrieve file by ID
- `delete()` - Soft delete + cleanup
- `get_versions()` - File version history

4. CreditService

- `pre_authorize()` - Freeze credits before execution
- `settle_transaction()` - Record actual cost
- `get_balance()` - User's available credits
- `get_transaction_history()` - Billing records
- `refund()` - Credit adjustment
- `deduct()` - Direct debit

5. ExportService

- `export_to_docx()` - Word document generation
- `export_to_pdf()` - PDF with formatting
- `export_to_txt()` - Plain text
- `export_to_markdown()` - Markdown format
- `get_export_history()` - Previous exports
- `download_export()` - Retrieve export file

6. MarketDataService

- `get_quote()` - Real-time price + bid/ask
- `get_ohlc()` - Open/High/Low/Close data
- `search_symbols()` - Symbol lookup + auto-complete
- `get_flash_news()` - Latest market news
- `get_economic_calendar()` - Upcoming events
- `search_news()` - Full-text news search

7. MarketHistoryStore

- `store_quote()` - Cache prices (Redis TTL: 2-10s)
- `get_history()` - Historical price data
- `get_trending()` - Trending symbols
- `clear_cache()` - Manual refresh

8. SessionFileEditService

- `track_edits()` - Log file modifications
- `get_diff()` - Compare versions
- `rollback()` - Revert to previous version
- `get_version_history()` - All edits timeline

9. ProviderFileService

- `upload_to_gcs()` - Google Cloud Storage
- `upload_to_minio()` - MinIO/S3 fallback
- `download()` - Retrieve from external storage
- `delete()` - Remove from cloud

10. AgentMetricsService

- `record_execution()` - Log agent run
- `get_metrics()` - Performance statistics
- `get_tool_usage()` - Which tools used most
- `get_cost_analytics()` - Spending by provider

11. AttachmentMetrics

- `track_attachment()` - Record file attachment
- `get_attachment_stats()` - Usage patterns
- `get_popular_formats()` - File type distribution

12. AttachmentRouting

- `route_file()` - Decide storage destination
- `check_quota()` - User storage limits
- `cleanup_old_files()` - Automatic pruning

13. MarkdownRenderer

- `render()` - Markdown to HTML
- `extract_code_blocks()` - Get executable code
- `extract_tables()` - Table parsing

14. AuthChallengeService

- `create_challenge()` - Generate verification code
- `verify_challenge()` - Check code validity
- `get_challenge_status()` - Current challenge state

15. TelemetryService (Implicit)

- `log_event()` - Track user actions
- `record_error()` - Error reporting
- `get_analytics()` - Usage analytics

MARKET INTELLIGENCE SYSTEM

Overview

A complete financial data platform integrated into Lumina, providing:

- Real-time stock/crypto quotes
- Historical OHLC data
- Flash news and alerts
- Economic calendar events
- News search and analysis

Implementation Details

```
# MarketDataService Architecture

class MarketDataService:
    """Real-time financial data provider."""

    def __init__(self):
        self.cache = RedisCache(ttl_seconds=2)  # Very short TTL for live data
        self.serper = SerperAPIClient()         # Primary source: Serper.dev
        self.fallback = PerplexityAI()         # Fallback for news

    async def get_quote(self, symbol: str) -> QuoteData:
        """Get current price, bid/ask, volume, change %."""
        key = f"quote:{symbol}"
        cached = await self.cache.get(key)
        if cached:
            return cached

        # Query Serper API (real-time data)
        data = await self.serper.get_quote(symbol)
```

```
        await self.cache.set(key, data, ttl=2)  # 2 second cache
        return data

    async def get_ohlc(
        self,
        symbol: str,
        interval: str,  # "1m", "5m", "1h", "1d", "1w", "1M", "1y"
        limit: int = 100
    ) -> List[OHLCBar]:
        """Get historical candlestick data."""
        key = f"ohlc:{symbol}:{interval}"
        cached = await self.cache.get(key)
        if cached:
            return cached[:limit]

        # Query historical data
        data = await self.serper.get_ohlc(symbol, interval, limit)
        await self.cache.set(key, data, ttl=300)  # 5 min cache for historical
        return data[:limit]

    async def get_flash_news(self, limit: int = 10) -> List[NewsItem]:
        """Latest market news with dates and sentiment."""
        key = "flash_news"
        cached = await self.cache.get(key)
        if cached:
            return cached[:limit]

        # Aggregate from multiple sources
        news = await self.serper.get_latest_news(limit=20)

        # Add sentiment analysis (optional)
        for item in news:
            item.sentiment = await self.analyze_sentiment(item.title)

        await self.cache.set(key, news, ttl=10)  # 10 second cache
        return news[:limit]

    async def search_news(self, query: str, symbol: Optional[str] = None) -> List[NewsItem]:
        """Full-text search across financial news."""
        # Multi-source search (Serper + Perplexity)
        results = await asyncio.gather(
            self.serper.search_news(query),
            self.fallback.search_news(query, symbol)
        )
        return sorted(
            [*results[0], *results[1]],
            key=lambda x: x.date,
            reverse=True
        )[:50]

    async def get_economic_calendar(self) -> List[EconomicEvent]:
        """Upcoming economic events with impact levels."""
        key = "economic_calendar"
        cached = await self.cache.get(key)
        if not cached or cached[0].date < datetime.now():
            # Re-fetch if calendar expired
            data = await self.serper.get_economic_calendar()
            await self.cache.set(key, data, ttl=3600)  # 1 hour cache
            return data
        return cached
```

Market Data Models

```
# PostgreSQL: market_history table
CREATE TABLE market_history (
    id UUID PRIMARY KEY,
    symbol VARCHAR(20) NOT NULL,
    quote_time TIMESTAMP NOT NULL,
    price DECIMAL(12, 2),
    bid_price DECIMAL(12, 2),
    ask_price DECIMAL(12, 2),
    volume BIGINT,
    change_percent DECIMAL(5, 2),
    created_at TIMESTAMP DEFAULT NOW(),

    INDEX (symbol, quote_time DESC)
);

# Time-series storage in Redis for quick access
Key: market:quote:{SYMBOL}
Value: {price, bid, ask, volume, time}
TTL: 2-10 seconds
```

Market Data Usage in Chat

When users ask market-related questions in chat, the agent can:

```
# In agent.py - Available tools:
{
    "tool": "market_quote",
    "description": "Get real-time stock/crypto price",
    "parameters": {
        "symbol": "AAPL|BTC|SPY|etc.",
        "include_history": true  # Include 24h high/low
    }
}

# Agent output example:
User: "What's Apple stock price and recent trends?"

Agent:
1. Thinks: "Need current quote + historical trend"
2. Calls: tool.market_quote(symbol="AAPL", include_history=true)
3. Gets: {price: 182.45, bid: 182.42, ask: 182.47, change: +1.23%, ...}
4. Responds: "Apple (AAPL) is trading at $182.45, up 1.23% today..."
```

RESEARCH & LEARNING PLATFORM

Overview

Located in `my_project/` and `scripts/` directories. A distributed research platform with:

- Web search integration (Serper + Perplexity + custom crawling)
- Academic search (scholar.google.com integration)
- Media compression (image, video, audio)
- Learning pack generation (quizzes, summaries)
- Async task processing (Celery + RabbitMQ)

Architecture

```
Frontend (Next.js)
    ↓
API: POST /api/v1/research/jobs (FastAPI)
    ↓
FastAPI queues task to Redis
    ↓
Celery Worker picks up task
    ↓
Flask Tools Service (tools_web:5000)
    ├─ ResearchService (web search, academic search)
    ├─ LearningPackService (quiz generation)
    ├─ MediaCompressService (image/video compression)
    └─ Results stored back to database
    ↓
Frontend polls GET /api/v1/research/jobs/:id
    ↓
Browser displays results
```

ResearchService Implementation

```python
# my_project/research_service.py

class ResearchService:
    """Multi-source research engine."""

    async def web_search(
        self,
        query: str,
        num_results: int = 10,
        include_snippets: bool = True
    ) -> List[SearchResult]:
        """Search web using multiple providers."""

        # Primary: Serper API (structured data)
        serper_results = await self.serper_client.search(query, num_results)

        # Fallback: Perplexity AI (semantic search with sources)
        perplexity_results = await self.perplexity_client.search(query, num_results)

        # Tertiary: Firecrawl (full-page markdown extraction)
        crawled_results = await self.firecrawl_client.crawl_urls(
            [r.url for r in serper_results[:5]]
        )

        # Merge results with deduplication
        merged = {}
        for result in serper_results + perplexity_results:
            if result.url not in merged:
                merged[result.url] = {
                    "title": result.title,
                    "snippet": result.snippet,
                    "url": result.url,
                    "source": "serper|perplexity",
                    "content": None
                }

        # Enrich with full content from Firecrawl
        for crawled in crawled_results:
            if crawled.url in merged:
                merged[crawled.url]["content"] = crawled.markdown

        return list(merged.values())

    async def academic_search(
        self,
        query: str,
        num_results: int = 10
    ) -> List[AcademicResult]:
        """Search Google Scholar."""

        # Uses scholarly library + caching
        results = await self.scholar_client.search(query)

        return [
            {
                "title": r.title,
                "authors": r.authors,
                "year": r.year,
                "citations": r.citations,
                "url": r.url,
                "abstract": r.abstract
            }
            for r in results[:num_results]
        ]
```

LearningPackService

```
class LearningPackService:
    """Generate educational content from research."""

    async def generate_quiz(
        self,
        content: str,
        num_questions: int = 5,
        difficulty: str = "medium"
    ) -> List[QuizQuestion]:
        """Generate quiz from text content."""

        # Use LLM to extract key concepts
        concepts = await self.llm.extract_concepts(content)

        # Generate multiple choice questions
        questions = []
        for concept in concepts[:num_questions]:
            q = await self.llm.generate_question(
                concept,
                content,
                difficulty=difficulty
            )
            questions.append(q)

        return questions

    async def generate_summary(
        self,
        content: str,
        summary_type: str = "bullet"  # bullet, paragraph, outline
    ) -> str:
        """Generate summary from content."""

        if summary_type == "bullet":
            return await self.llm.summarize_bullets(content)
        elif summary_type == "paragraph":
            return await self.llm.summarize_paragraph(content)
        else:
            return await self.llm.generate_outline(content)
```

MediaCompressService

```
class MediaCompressService:
    """Media compression with tiered quotas."""

    COMPRESSION_QUOTAS = {
        "free": {"image_gb": 1, "video_gb": 0.5, "audio_gb": 0.2},
        "pro": {"image_gb": 10, "video_gb": 5, "audio_gb": 2},
        "enterprise": {"image_gb": 100, "video_gb": 50, "audio_gb": 20}
    }

    async def compress_image(
        self,
        file_path: str,
        quality: int = 80,
        max_width: int = 1920
    ) -> bytes:
        """Compress image using PIL."""

        # Check quota
        user_tier = await self.get_user_tier()
        quota = self.COMPRESSION_QUOTAS[user_tier]
        used = await self.get_storage_used("image")

        if used >= quota["image_gb"] * 1024:
            raise QuotaExceededError()

        # Compress
        image = Image.open(file_path)
        image.thumbnail((max_width, max_width))

        output = BytesIO()
        image.save(output, format="JPEG", quality=quality, optimize=True)

        return output.getvalue()

    async def compress_video(
        self,
        file_path: str,
        preset: str = "medium"  # fast, medium, slow
    ) -> bytes:
        """Compress video using FFmpeg."""

        # Check quota
        quotas = self.COMPRESSION_QUOTAS[await self.get_user_tier()]
        if await self.get_storage_used("video") >= quotas["video_gb"] * 1024:
            raise QuotaExceededError()

        # Use FFmpeg for compression
        output = await self.ffmpeg.compress(
            file_path,
            preset=preset,
            codec="h264",
            bitrate="2000k"
        )

        return output
```

Celery Task Queue Integration

```python
# my_project/tasks.py

from celery import Celery

app = Celery('lumina_research')
app.config_from_object('celery_config')

@app.task(bind=True)
def research_web_async(self, query: str, job_id: UUID):
    """Async web research task."""
    try:
        # Update job status
        job = ResearchJob.get(job_id)
        job.status = "processing"
        job.save()

        # Run research
        service = ResearchService()
        results = asyncio.run(service.web_search(query))

        # Save results
        job.results = results
        job.status = "completed"
        job.completed_at = datetime.now()
        job.save()

    except Exception as e:
        job.status = "failed"
        job.error = str(e)
        job.save()

@app.task(bind=True)
def compress_media_async(self, file_id: UUID, compression_type: str):
    """Async media compression."""
    # Similar pattern...
```

MATLAB/SCIENTIFIC COMPUTING

Overview

Full MATLAB/Octave integration for scientific computing with:

- Code execution in sandboxed environment
- Advanced graphing (multiple chart types)
- Output visualization
- CJK font support (Chinese/Japanese/Korean)
- Automatic Octave fallback (open-source alternative)

MATLAB Executor Implementation

```python
# app/engine/matlab_executor.py

class MATLABExecutor:
    """Execute MATLAB/Octave code in isolated sandbox."""

    def __init__(self):
        self.octave_path = "/usr/bin/octave"  # Or MATLAB path
        self.graphics_toolkit = "gnuplot"
        self.sandbox_dir = None

    async def execute(
        self,
        code: str,
        session_id: UUID,
```

```
        session_id: str,
        timeout_seconds: int = 120
    ) -> ExecutionResult:
        """Execute MATLAB/Octave code."""

        # 1. Create session-specific sandbox
        self.sandbox_dir = f"/agent_sandboxes/{session_id}/matlab"
        os.makedirs(self.sandbox_dir, exist_ok=True)

        # 2. Sanitize code (remove dangerous commands)
        safe_code = self.sanitize_code(code)

        # 3. Prepare script file
        script_file = f"{self.sandbox_dir}/script.m"
        with open(script_file, 'w', encoding='utf-8') as f:
            f.write(f"""
% Set graphics properties
figure('Visible', 'off');
set(groot, 'defaultAxesTickLabelInterpreter', 'latex');
set(groot, 'defaultTextInterpreter', 'latex');

% Add CJK font support
set(groot, 'defaultAxesFontName', 'DejaVu Sans');

% User code
{safe_code}

% Save figures
savefig('{self.sandbox_dir}/output.png', '-r150');
close all;

% Exit cleanly
exit(0);
""")

        # 4. Execute with resource limits
        try:
            process = await asyncio.create_subprocess_exec(
                self.octave_path,
                "--no-gui",
                "--eval",
                f'run(\"{script_file}\")',
                stdout=asyncio.subprocess.PIPE,
                stderr=asyncio.subprocess.PIPE,
                cwd=self.sandbox_dir,
                # Resource limits
                limit=500 * 1024 * 1024,  # 500MB memory
            )

            # Wait with timeout
            stdout, stderr = await asyncio.wait_for(
                process.communicate(),
                timeout=timeout_seconds
            )

            # 5. Collect outputs
            outputs = {
                "stdout": stdout.decode('utf-8', errors='ignore'),
                "stderr": stderr.decode('utf-8', errors='ignore'),
                "figures": self.collect_figures(),
                "variables": self.collect_variables(),
            }

            return ExecutionResult(
                success=process.returncode == 0,
                output=outputs,
```

```
                runtime_ms=time.time() * 1000,
                sandbox_path=self.sandbox_dir
            )

        except asyncio.TimeoutError:
            return ExecutionResult(
                success=False,
                error="Execution timeout (120 seconds exceeded)"
            )

    def sanitize_code(self, code: str) -> str:
        """Remove dangerous MATLAB commands."""
        dangerous_patterns = [
            r"system\s*\(",      # system() calls
            r"!.*",              # Shell escape
            r"eval\s*\(",        # eval() execution
            r"system\(.*\)",     # system() function
            r"cd\s*\(",          # Change directory
            r"rmdir\s*\(",       # Remove directory
            r"delete\s*\(",      # Delete files
        ]

        for pattern in dangerous_patterns:
            code = re.sub(pattern, "% BLOCKED: " + pattern, code)

        return code

    def collect_figures(self) -> List[Figure]:
        """Extract generated figures."""
        figures = []
        for i, png_file in enumerate(glob.glob(f"{self.sandbox_dir}/*.png")):
            with open(png_file, 'rb') as f:
                base64_data = base64.b64encode(f.read()).decode()
                figures.append({
                    "index": i,
                    "filename": os.path.basename(png_file),
                    "data": base64_data,
                    "size_kb": os.path.getsize(png_file) / 1024
                })
        return figures

    def collect_variables(self) -> dict:
        """Extract workspace variables."""
        # Use Octave's '--eval' to dump workspace
        var_file = f"{self.sandbox_dir}/variables.json"
        # Octave writes variables to JSON file
        # Read and parse
        if os.path.exists(var_file):
            with open(var_file) as f:
                return json.load(f)
        return {}
```

MATLAB in Agent System

When agent executes MATLAB tool:

```
# In agent executor

tool_call = {
    "tool": "execute_matlab",
    "parameters": {
        "code": """
        % Linear regression example
        x = 1:10;
        y = 2*x + randn(1,10);
        p = polyfit(x, y, 1);
        plot(x, y, 'o', x, polyval(p, x));
        xlabel('X');
        ylabel('Y');
        title('Linear Regression');
        grid on;
        """
    }
}

# Execution flow:
1. Permission check: ALLOW (user has execute_matlab permission)
2. Run MATLABExecutor.execute(code)
3. Get back: {stdout, stderr, figures: [base64], variables: {...}}
4. Agent includes in next message:
   "Executed linear regression. Generated figure showing fit."
5. Frontend displays figure inline
```

FILE MANAGEMENT & VERSIONING

Overview

Complete file lifecycle management with:

- Deduplication via SHA256 hashing
- Version history and rollback
- Diff viewing
- Session-level file isolation
- Multi-provider storage (local, S3, GCS)

File Models & Schema

```
# Database schema

class File(Base):
    """Master file record."""
    __tablename__ = "files"

    id: UUID = Column(UUID, primary_key=True)
    user_id: UUID = Column(UUID, ForeignKey("users.id"))
    filename: str = Column(String)
    content_hash: str = Column(String, unique=True)  # SHA256
    mime_type: str = Column(String)
    size_bytes: int = Column(BigInteger)
    created_at: datetime = Column(DateTime, default=now)

    # Relationships
    session_files = relationship("SessionFile", back_populates="file")


class SessionFile(Base):
    """File attached to a session (junction)."""
    __tablename__ = "session_files"

    id: UUID = Column(UUID, primary_key=True)
    session_id: UUID = Column(UUID, ForeignKey("sessions.id"))
    file_id: UUID = Column(UUID, ForeignKey("files.id"))
    attached_message_id: UUID = Column(UUID, ForeignKey("messages.id"), nullable=True)
    created_at: datetime = Column(DateTime, default=now)

    # Relationships
    file = relationship("File", back_populates="session_files")
    versions = relationship("SessionFileVersion", back_populates="session_file")


class SessionFileVersion(Base):
    """Version history for each session_file."""
    __tablename__ = "session_file_versions"

    id: UUID = Column(UUID, primary_key=True)
    session_file_id: UUID = Column(UUID, ForeignKey("session_files.id"))
    version_number: int = Column(Integer)
    content: str = Column(Text)  # Or LargeBinary for binary files
    content_hash: str = Column(String)  # To detect changes
    edited_by: str = Column(String, nullable=True)  # Editor name
    edit_reason: str = Column(String, nullable=True)  # Why was it edited?
    created_at: datetime = Column(DateTime, default=now)

    # Relationships
    session_file = relationship("SessionFile", back_populates="versions")
```

File Service Implementation

```
class FileService:
    """File upload, deduplication, versioning."""

    async def upload(
        self,
        user_id: UUID,
        file: UploadFile,
        session_id: Optional[UUID] = None
    ) -> File:
        """Upload file with deduplication."""

        # 1. Read file content
        content = await file.read()
```

```
    # 2. Calculate SHA256 hash
    content_hash = hashlib.sha256(content).hexdigest()

    # 3. Check if already exists
    existing = await File.query.filter(
        File.content_hash == content_hash
    ).first()

    if existing:
        # Deduplication: file already exists
        print(f"✓ File deduplicated (hash match)")
        file_obj = existing
    else:
        # New file: store to disk/cloud
        file_obj = File(
            id=uuid4(),
            user_id=user_id,
            filename=file.filename,
            content_hash=content_hash,
            mime_type=file.content_type,
            size_bytes=len(content)
        )

        # Store content
        storage_path = await self.store_content(file_obj.id, content)
        file_obj.storage_path = storage_path

        await db.add(file_obj)
        await db.commit()

    # 4. Link to session if provided
    if session_id:
        session_file = SessionFile(
            id=uuid4(),
            session_id=session_id,
            file_id=file_obj.id,
            created_at=datetime.now()
        )
        await db.add(session_file)
        await db.commit()

    return file_obj

async def get_versions(self, session_file_id: UUID) -> List[SessionFileVersion]:
    """Get version history."""
    return await SessionFileVersion.query.filter(
        SessionFileVersion.session_file_id == session_file_id
    ).order_by(SessionFileVersion.version_number.desc()).all()

async def rollback(
    self,
    session_file_id: UUID,
    version_number: int
) -> SessionFileVersion:
    """Restore to specific version."""

    # Get target version
    target = await SessionFileVersion.query.filter(
        SessionFileVersion.session_file_id == session_file_id,
        SessionFileVersion.version_number == version_number
    ).first()

    if not target:
        raise VersionNotFound()
```

```
        # Create new version (don't overwrite history)
        new_version = SessionFileVersion(
            id=uuid4(),
            session_file_id=session_file_id,
            version_number=await self.get_next_version_number(session_file_id),
            content=target.content,
            content_hash=target.content_hash,
            edit_reason=f"Rollback to version {version_number}",
            created_at=datetime.now()
        )

        await db.add(new_version)
        await db.commit()

        return new_version

    async def get_diff(
        self,
        session_file_id: UUID,
        version_a: int,
        version_b: int
    ) -> DiffResult:
        """Compare two versions."""

        v_a = await self.get_version(session_file_id, version_a)
        v_b = await self.get_version(session_file_id, version_b)

        if not v_a or not v_b:
            raise VersionNotFound()

        # Use difflib for comparison
        diff = difflib.unified_diff(
            v_a.content.splitlines(keepends=True),
            v_b.content.splitlines(keepends=True),
            fromfile=f"v{version_a}",
            tofile=f"v{version_b}"
        )

        return DiffResult(
            version_a=version_a,
            version_b=version_b,
            diff=''.join(diff),
            a_timestamp=v_a.created_at,
            b_timestamp=v_b.created_at
        )
```

Storage Routing

```
class ProviderFileService:
    """Route files to optimal storage."""

    async def store_content(
        self,
        file_id: UUID,
        content: bytes
    ) -> str:
        """Store to local or cloud based on size."""

        size_mb = len(content) / (1024 * 1024)

        if size_mb < 50:
            # Small files: local disk
            path = f"./uploads/{file_id}"
            with open(path, 'wb') as f:
                f.write(content)
            return f"local://{path}"

        elif size_mb < 1000:
            # Medium files: MinIO (S3-compatible)
            key = f"files/{file_id}"
            await self.minio_client.put_object(
                bucket_name="lumina-files",
                object_name=key,
                data=BytesIO(content),
                length=len(content)
            )
            return f"s3://{key}"

        else:
            # Large files: Google Cloud Storage
            blob_name = f"archive/{file_id}"
            await self.gcs_client.upload_blob(
                bucket_name="lumina-archive",
                blob_name=blob_name,
                data=content
            )
            return f"gcs://{blob_name}"
```

EXPORT SYSTEM

Overview

Multi-format export functionality for session conversations:

```
Session → Markdown (intermediate) → DOCX/PDF/TXT
```

Export Service

```
class ExportService:
    """Export sessions to various formats."""

    async def export_session(
        self,
        session_id: UUID,
        format: str,  # "docx", "pdf", "txt", "markdown"
        include_thinking: bool = False,
        include_tool_calls: bool = True
    ) -> ExportFile:
        """Export session conversation."""
```

```
    # 1. Get all session messages
    session = await Session.get(session_id)
    messages = await Message.query.filter(
        Message.session_id == session_id
    ).order_by(Message.created_at).all()

    # 2. Convert to markdown (universal format)
    markdown = await self.to_markdown(messages, include_thinking, include_tool_calls)

    # 3. Convert to target format
    if format == "markdown":
        content = markdown
    elif format == "txt":
        content = await self.markdown_to_txt(markdown)
    elif format == "docx":
        content = await self.markdown_to_docx(markdown)
    elif format == "pdf":
        content = await self.markdown_to_pdf(markdown)
    else:
        raise InvalidExportFormat()

    # 4. Store export
    export_file = ExportFile(
        id=uuid4(),
        session_id=session_id,
        format=format,
        filename=f"session_{session_id}_{datetime.now().strftime('%Y%m%d_%H%M%S')}.{format}",
        content=content,
        created_at=datetime.now()
    )

    await db.add(export_file)
    await db.commit()

    return export_file

async def to_markdown(
    self,
    messages: List[Message],
    include_thinking: bool,
    include_tool_calls: bool
) -> str:
    """Convert messages to markdown."""

    md_lines = [
        f"# Session Export",
        f"**Date**: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}",
        "",
    ]

    for msg in messages:
        # Render message
        if msg.role == "user":
            md_lines.append(f"## User:")
            md_lines.append(msg.content)
        else:
            md_lines.append(f"## Assistant:")
            md_lines.append(msg.content)

        # Add metadata
        if include_thinking and msg.metadata.get('thinking_process'):
            md_lines.append(f"\n*Thinking*:")
            md_lines.append(f"```\n{msg.metadata['thinking_process']}\n```")

        if include_tool_calls and msg.metadata.get('tool_calls'):
```

```
                md_lines.append("\n*Tool Calls*:")
                for tool_call in msg.metadata['tool_calls']:
                    md_lines.append(f"- `{tool_call['tool']}`")

            md_lines.append("")

        return "\n".join(md_lines)

    async def markdown_to_docx(self, markdown: str) -> bytes:
        """Convert markdown to DOCX using python-docx."""

        doc = Document()
        doc.add_heading('Session Export', 0)

        # Parse markdown and add to document
        for line in markdown.split('\n'):
            if line.startswith('# '):
                doc.add_heading(line[2:], 1)
            elif line.startswith('## '):
                doc.add_heading(line[3:], 2)
            elif line.startswith('```'):
                # Code block - skip markers
                continue
            elif line.strip():
                doc.add_paragraph(line)

        # Save to bytes
        output = BytesIO()
        doc.save(output)
        return output.getvalue()

    async def markdown_to_pdf(self, markdown: str) -> bytes:
        """Convert markdown to PDF using WeasyPrint."""

        # Convert markdown to HTML first
        html = markdown.replace('\n\n', '</p><p>')

        # Render to PDF
        from weasyprint import HTML, CSS

        pdf_bytes = HTML(string=f"<html><body>{html}</body></html>").write_pdf()

        return pdf_bytes
```

ASYNC JOB PROCESSING

Celery + RabbitMQ Architecture

Task Definition

```python
# app/worker/tasks.py

from celery import Celery, Task

app = Celery(
    'lumina',
    broker='amqp://guest:guest@rabbitmq:5672//',
    backend='redis://redis:6379/0'
)

@app.task(bind=True, max_retries=3)
def export_session_async(self, session_id: str, format: str, user_id: str):
    """Async export task."""
    try:
        # Mark as processing
        job = ResearchJob.get(session_id)
        job.status = "processing"
        job.save()

        # Run export
        export_service = ExportService()
        result = asyncio.run(export_service.export_session(
            UUID(session_id),
            format
        ))

        # Mark complete
        job.status = "completed"
        job.result_url = f"/downloads/{result.id}"
        job.save()

    except Exception as exc:
        # Retry with exponential backoff
        self.retry(exc=exc, countdown=60)

@app.task(bind=True)
def research_job_async(self, query: str, job_id: str):
    """Async research job."""
    try:
        job = ResearchJob.get(job_id)
        job.status = "processing"
        job.save()

        service = ResearchService()
        results = asyncio.run(service.web_search(query))

        job.status = "completed"
        job.results = results
        job.save()

    except Exception as e:
        job.status = "failed"
        job.error = str(e)
        job.save()
```

Frontend Integration

```
// lumina_web/hooks/useAsyncJob.ts

export const useAsyncJob = (jobId: string) => {
  const [status, setStatus] = useState<'pending' | 'processing' | 'completed' | 'failed'>('pending')
  const [result, setResult] = useState(null)
  const [error, setError] = useState(null)

  useEffect(() => {
    const interval = setInterval(async () => {
      const response = await fetch(`/api/v1/research/jobs/${jobId}`)
      const job = await response.json()

      setStatus(job.status)
      if (job.status === 'completed') {
        setResult(job.result)
        clearInterval(interval)
      }
      if (job.status === 'failed') {
        setError(job.error)
        clearInterval(interval)
      }
    }, 1000)

    return () => clearInterval(interval)
  }, [jobId])

  return { status, result, error }
}
```

WEB SCRAPING INTEGRATION

Firecrawl Self-Hosted

```
Frontend Request
    ↓
FastAPI: POST /api/v1/research/jobs
    ↓
Celery Task Queued
    ↓
Celery Worker
    ├─ Call Firecrawl API (localhost:3002)
    ├─ Pass URL list
    ├─ Get markdown-formatted content
    ├─ Store in database
    └─ Signal completion
    ↓
Frontend Polls for Results
    ↓
Display extracted content
```

Implementation

```
class FirecrawlService:
    """Self-hosted web scraping."""

    def __init__(self):
        self.base_url = "http://firecrawl:3002"

    async def crawl_url(self, url: str) -> ScrapedContent:
        """Scrape URL and return markdown."""

        async with httpx.AsyncClient() as client:
            response = await client.post(
                f"{self.base_url}/v0/crawl",
                json={
                    "url": url,
                    "formats": ["markdown"],
                    "onlyMainContent": True,
                    "waitFor": 2000  # Wait for JS rendering
                },
                headers={"Content-Type": "application/json"}
            )

        result = response.json()

        return ScrapedContent(
            url=url,
            markdown=result['data']['markdown'],
            title=result['data'].get('title', ''),
            metadata=result['data'].get('metadata', {})
        )

    async def crawl_urls_batch(self, urls: List[str]) -> List[ScrapedContent]:
        """Crawl multiple URLs (batched for efficiency)."""

        results = []
        for url in urls:
            try:
                content = await self.crawl_url(url)
                results.append(content)
            except Exception as e:
                print(f"Failed to crawl {url}: {e}")
                continue

        return results
```

MEDIA PROCESSING

Image Compression

```
class ImageCompressionService:
    """Image compression with format conversion."""

    async def compress(
        self,
        input_path: str,
        output_format: str = "jpeg",
        quality: int = 80,
        max_width: int = 1920
    ) -> bytes:
        """Compress and convert image."""

        # Open image
        image = Image.open(input_path)

        # Resize if too large
        if image.width > max_width:
            ratio = max_width / image.width
            new_height = int(image.height * ratio)
            image = image.resize((max_width, new_height), Image.Resampling.LANCZOS)

        # Convert to target format
        output = BytesIO()

        if output_format.lower() == "jpeg":
            image = image.convert("RGB")
            image.save(output, format="JPEG", quality=quality, optimize=True)
        elif output_format.lower() == "webp":
            image.save(output, format="WebP", quality=quality, method=6)
        elif output_format.lower() == "png":
            image.save(output, format="PNG", optimize=True)

        return output.getvalue()
```

Video Compression

```
class VideoCompressionService:
    """Video compression using FFmpeg."""

    async def compress(
        self,
        input_path: str,
        bitrate: str = "2000k",
        max_width: int = 1280
    ) -> bytes:
        """Compress video."""

        output_path = f"/tmp/output_{uuid4()}.mp4"

        # FFmpeg command
        cmd = [
            "ffmpeg",
            "-i", input_path,
            "-codec:v", "libx264",
            "-preset", "medium",
            "-b:v", bitrate,
            "-vf", f"scale=min({max_width}\\,iw):-2",
            "-codec:a", "aac",
            "-b:a", "128k",
            output_path
        ]

        process = await asyncio.create_subprocess_exec(*cmd)
        await process.wait()

        with open(output_path, 'rb') as f:
            result = f.read()

        os.remove(output_path)

        return result
```

Audio Compression & Transcription

```
class AudioService:
    """Audio processing."""

    async def compress(
        self,
        input_path: str,
        bitrate: str = "128k"
    ) -> bytes:
        """Compress audio (MP3)."""

        output_path = f"/tmp/output_{uuid4()}.mp3"

        cmd = [
            "ffmpeg",
            "-i", input_path,
            "-b:a", bitrate,
            output_path
        ]

        process = await asyncio.create_subprocess_exec(*cmd)
        await process.wait()

        with open(output_path, 'rb') as f:
            result = f.read()

        os.remove(output_path)
        return result

    async def transcribe(self, audio_path: str) -> str:
        """Transcribe audio using Groq Whisper."""

        with open(audio_path, 'rb') as f:
            response = await self.groq_client.audio.transcriptions.create(
                file=f,
                model="whisper-large-v3-turbo"
            )

        return response.text
```

AGENT EXECUTOR DEEP DIVE

ReAct Loop with 30 Iterations

```
class AgentExecutor:
    """ReAct (Reasoning + Acting) loop executor."""

    MAX_ITERATIONS = 30
    MAX_RUNTIME_SECONDS = 600
    MAX_PARALLEL_TOOLS = 3

    async def execute(
        self,
        messages: List[Message],
        user_id: UUID,
        session_id: UUID,
        tools: Optional[List[Tool]] = None
    ) -> AsyncIterator[str]:
        """Execute agent in ReAct loop."""

        context = {
            "iteration": 0,
            "start_time": time.time(),
```

```
        "messages": messages,
        "observations": [],
        "user_id": user_id,
        "session_id": session_id,
        "cost_tracker": CostTracker(user_id)
    }

    while context["iteration"] < self.MAX_ITERATIONS:
        # 1. THINKING: Get LLM response
        yield f"event: thinking\ndata: Iteration {context['iteration'] + 1}\n\n"

        thinking_response = await self.llm_gateway.route(
            request=ChatRequest(
                messages=context["messages"],
                tools=tools,
                system_prompt=AGENT_SYSTEM_PROMPT
            )
        )

        yield f"event: thinking_result\ndata: {thinking_response}\n\n"

        # 2. ACTION SELECTION: Parse tool calls
        tool_calls = self.parse_tool_calls(thinking_response)

        if not tool_calls:
            # No more tools -> Final answer
            yield f"event: done\ndata: {thinking_response}\n\n"
            break

        # 3. PERMISSION CHECK: Verify each tool
        approved_tools = []
        for tool_call in tool_calls:
            permission = await self.permission_model.check(
                user_id,
                tool_call.name,
                tool_call.parameters
            )

            if permission == PermissionType.ALLOW:
                approved_tools.append(tool_call)
            elif permission == PermissionType.ASK:
                # Prompt user for approval
                yield f"event: permission_ask\ndata: {tool_call.name}\n\n"
                # Wait for user approval...
                approved_tools.append(tool_call)
            # else: PermissionType.DENY -> skip tool

        # 4. EXECUTION: Run tools in parallel (max 3)
        results = await self.execute_tools_parallel(
            approved_tools,
            context,
            max_parallel=self.MAX_PARALLEL_TOOLS
        )

        for result in results:
            yield f"event: tool_result\ndata: {result}\n\n"
            context["observations"].append(result)

        # 5. LOOP CONDITION CHECK
        context["iteration"] += 1

        if time.time() - context["start_time"] > self.MAX_RUNTIME_SECONDS:
            yield f"event: timeout\ndata: Execution exceeded {self.MAX_RUNTIME_SECONDS}s\n\n"
            break
```

```
        # Save execution metrics
        await self.save_metrics(context)

    async def execute_tools_parallel(
        self,
        tool_calls: List[ToolCall],
        context: dict,
        max_parallel: int
    ) -> List[ToolResult]:
        """Execute multiple tools in parallel."""

        results = []

        # Batch into groups of max_parallel
        for i in range(0, len(tool_calls), max_parallel):
            batch = tool_calls[i:i+max_parallel]

            tasks = [
                self.execute_tool(tool_call, context)
                for tool_call in batch
            ]

            batch_results = await asyncio.gather(*tasks, return_exceptions=True)
            results.extend(batch_results)

        return results
```

Tool Execution with Sandboxing

```
async def execute_tool(self, tool_call: ToolCall, context: dict) -> ToolResult:
    """Execute single tool with security checks."""

    tool_name = tool_call.name
    parameters = tool_call.parameters
    session_id = context["session_id"]
    user_id = context["user_id"]

    try:
        # 1. Rate limiting check
        rate_limited = await self.rate_limiter.check(
            user_id,
            tool_name
        )
        if rate_limited:
            return ToolResult(
                tool=tool_name,
                success=False,
                error="Rate limit exceeded for this tool"
            )

        # 2. Credit pre-authorization
        estimated_cost = await self.estimate_tool_cost(tool_name, parameters)
        auth_token = await self.credit_service.pre_authorize(
            user_id,
            estimated_cost
        )

        # 3. Execute tool
        if tool_name == "execute_python":
            result = await self.python_executor.execute(
                code=parameters["code"],
                session_id=session_id,
                sandbox_dir=f"/agent_sandboxes/{session_id}"
            )
```

```
        elif tool_name == "execute_matlab":
            result = await self.matlab_executor.execute(
                code=parameters["code"],
                session_id=session_id
            )

        elif tool_name == "market_quote":
            result = await self.market_service.get_quote(
                symbol=parameters["symbol"],
                include_history=parameters.get("include_history", False)
            )

        elif tool_name == "web_search":
            result = await self.research_service.web_search(
                query=parameters["query"],
                num_results=parameters.get("num_results", 10)
            )

        # ... more tools

        # 4. Cost settlement
        actual_cost = await self.measure_tool_cost(tool_name, result)
        await self.credit_service.settle_transaction(auth_token, actual_cost)

        # 5. Result storage (compaction if large)
        if len(str(result)) > 10_000_000:  # 10MB
            stored_result = await self.store_large_result(result, session_id)
            result_preview = self.create_preview(result)
        else:
            stored_result = result
            result_preview = result

        return ToolResult(
            tool=tool_name,
            success=True,
            output=result_preview,
            full_output_id=stored_result.id if isinstance(stored_result, StoredResult) else None,
            cost=actual_cost
        )

    except Exception as e:
        return ToolResult(
            tool=tool_name,
            success=False,
            error=str(e)
        )
```

LLM MULTI-PROVIDER GATEWAY

Provider Abstraction

```
class LLMProvider(ABC):
    """Abstract interface for all LLM providers."""

    @abstractmethod
    async def chat_completion(
        self,
        messages: List[Message],
        model: str,
        tools: Optional[List[Tool]] = None,
        temperature: float = 0.7
    ) -> AsyncIterator[str]:
        """Stream chat completion."""
        pass

    @abstractmethod
    async def count_tokens(self, text: str, model: str) -> int:
        """Count tokens for cost calculation."""
        pass


class OpenAIProvider(LLMProvider):
    """OpenAI GPT-4, GPT-3.5, etc."""

    async def chat_completion(self, ...):
        async with aiohttp.ClientSession() as session:
            async with session.post(
                "https://api.openai.com/v1/chat/completions",
                json={...},
                headers={"Authorization": f"Bearer {self.api_key}"}
            ) as resp:
                async for chunk in resp.content.iter_chunked(8192):
                    yield chunk.decode()


class AnthropicProvider(LLMProvider):
    """Anthropic Claude."""

    async def chat_completion(self, ...):
        # Similar implementation for Anthropic


# ... GoogleProvider, DeepSeekProvider, etc.
```

Gateway Router with Fallback

```
class LLMGateway:
    """Multi-provider router with automatic fallback."""

    def __init__(self):
        self.providers = {
            "openai": OpenAIProvider(),
            "anthropic": AnthropicProvider(),
            "google": GoogleProvider(),
            "deepseek": DeepSeekProvider(),
        }

        self.provider_order = {
            "reasoning": ["openai", "anthropic", "google"],
            "speed": ["openai", "deepseek", "google"],
            "cost": ["deepseek", "google", "openai"],
        }

    async def route(
        self,
```

```
        request: ChatRequest,
        user_preferences: UserPreferences,
        strategy: str = "reasoning"  # reasoning, speed, cost
    ) -> AsyncIterator[str]:
        """Route to best provider with fallback."""

        provider_order = user_preferences.model_order or self.provider_order[strategy]

        for provider_name in provider_order:
            if provider_name not in self.providers:
                continue

            try:
                provider = self.providers[provider_name]

                # Convert request to provider format
                native_request = self.normalize_request(request, provider_name)

                # Stream response
                async for token in provider.chat_completion(**native_request):
                    yield token

                return  # Success!

            except RateLimitError:
                print(f"{provider_name} rate limited, trying next...")
                continue

            except ProviderError as e:
                print(f"{provider_name} error: {e}, trying next...")
                continue

        raise AllProvidersExhaustedError("All LLM providers failed")

    def normalize_request(self, request: ChatRequest, provider: str) -> dict:
        """Convert generic request to provider format."""

        base = {
            "messages": request.messages,
            "temperature": request.temperature,
            "max_tokens": request.max_tokens,
        }

        # Select model for provider
        base["model"] = self.select_model(request.task_type, provider)

        # Normalize tool format
        if request.tools:
            base["tools"] = self.normalize_tools(request.tools, provider)

        return base

    def normalize_tools(self, tools: List[Tool], provider: str) -> List[dict]:
        """Convert tools to provider-specific format."""

        if provider == "openai":
            return [{"type": "function", "function": tool.to_dict()} for tool in tools]
        elif provider == "anthropic":
            return [tool.to_anthropic_format() for tool in tools]
        elif provider == "google":
            return [tool.to_google_format() for tool in tools]
        else:
            return [tool.to_dict() for tool in tools]
```

DATABASE SCHEMA (ALL 12 MODELS)

```
 -- 1. USERS & AUTH
CREATE TABLE users (
    id UUID PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255),
    full_name VARCHAR(255),
    avatar_url VARCHAR(500),
    credits DECIMAL(10, 2) DEFAULT 0,
    tier VARCHAR(20) DEFAULT 'free',  -- free, pro, enterprise
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW(),
    last_login TIMESTAMP,
    is_active BOOLEAN DEFAULT TRUE,

    INDEX idx_email (email)
);

-- 2. SESSIONS
CREATE TABLE sessions (
    id UUID PRIMARY KEY,
    user_id UUID NOT NULL REFERENCES users(id),
    title VARCHAR(500),
    description TEXT,
    parent_session_id UUID REFERENCES sessions(id),  -- For branching
    parent_message_id UUID,  -- Branch point
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW(),
    archived_at TIMESTAMP,

    FOREIGN KEY (user_id) REFERENCES users(id),
    INDEX idx_user_id (user_id),
    INDEX idx_parent (parent_session_id)
);

-- 3. MESSAGES
CREATE TABLE messages (
    id UUID PRIMARY KEY,
    session_id UUID NOT NULL REFERENCES sessions(id),
    role VARCHAR(20) NOT NULL,  -- user, assistant
    content TEXT NOT NULL,
    metadata JSONB,  -- {tokens, cost, thinking_process, tool_calls}
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW(),
    is_edited BOOLEAN DEFAULT FALSE,

    FOREIGN KEY (session_id) REFERENCES sessions(id),
    INDEX idx_session_id (session_id),
    INDEX idx_created_at (created_at)
);

-- 4. FILES
CREATE TABLE files (
    id UUID PRIMARY KEY,
    user_id UUID NOT NULL REFERENCES users(id),
    filename VARCHAR(255),
    content_hash VARCHAR(64) UNIQUE NOT NULL,  -- SHA256
    mime_type VARCHAR(100),
    size_bytes BIGINT,
    storage_path VARCHAR(500),  -- local://, s3://, gcs://
    created_at TIMESTAMP DEFAULT NOW(),

    FOREIGN KEY (user_id) REFERENCES users(id),
    INDEX idx_content_hash (content_hash)
```

```
    INDEX idx_content_hash (content_hash),
    INDEX idx_user_id (user_id)
);

-- 5. SESSION_FILES (many-to-many)
CREATE TABLE session_files (
    id UUID PRIMARY KEY,
    session_id UUID NOT NULL REFERENCES sessions(id),
    file_id UUID NOT NULL REFERENCES files(id),
    attached_message_id UUID REFERENCES messages(id),
    created_at TIMESTAMP DEFAULT NOW(),

    FOREIGN KEY (session_id) REFERENCES sessions(id),
    FOREIGN KEY (file_id) REFERENCES files(id),
    INDEX idx_session_id (session_id),
    UNIQUE (session_id, file_id)
);

-- 6. SESSION_FILE_VERSIONS
CREATE TABLE session_file_versions (
    id UUID PRIMARY KEY,
    session_file_id UUID NOT NULL REFERENCES session_files(id),
    version_number INT NOT NULL,
    content MEDIUMTEXT,
    content_hash VARCHAR(64),
    edited_by VARCHAR(100),
    edit_reason VARCHAR(500),
    created_at TIMESTAMP DEFAULT NOW(),

    FOREIGN KEY (session_file_id) REFERENCES session_files(id),
    INDEX idx_session_file_id (session_file_id),
    UNIQUE (session_file_id, version_number)
);

-- 7. CREDIT_TRANSACTIONS
CREATE TABLE credit_transactions (
    id UUID PRIMARY KEY,
    user_id UUID NOT NULL REFERENCES users(id),
    session_id UUID REFERENCES sessions(id),
    message_id UUID REFERENCES messages(id),
    amount DECIMAL(10, 4),
    reason VARCHAR(100),  -- llm_call, code_execution, etc.
    provider VARCHAR(50),  -- openai, anthropic, etc.
    model VARCHAR(100),
    input_tokens INT,
    output_tokens INT,
    status VARCHAR(20),  -- pending, settled, failed
    created_at TIMESTAMP DEFAULT NOW(),
    settled_at TIMESTAMP,

    FOREIGN KEY (user_id) REFERENCES users(id),
    INDEX idx_user_id (user_id),
    INDEX idx_created_at (created_at)
);

-- 8. AUTH_CHALLENGES
CREATE TABLE auth_challenges (
    id UUID PRIMARY KEY,
    user_id UUID REFERENCES users(id),
    email VARCHAR(255),
    challenge_type VARCHAR(50),  -- email_verification, password_reset
    code VARCHAR(10),
    attempts INT DEFAULT 0,
    max_attempts INT DEFAULT 5,
    created_at TIMESTAMP DEFAULT NOW(),
    expires_at TIMESTAMP,
```

```
    verified_at TIMESTAMP,

    INDEX idx_email (email),
    INDEX idx_expires_at (expires_at)
);

-- 9. AGENT_METRICS
CREATE TABLE agent_metrics (
    id UUID PRIMARY KEY,
    session_id UUID NOT NULL REFERENCES sessions(id),
    user_id UUID NOT NULL REFERENCES users(id),
    iteration_count INT,
    total_runtime_ms BIGINT,
    tools_used JSONB,  -- ["execute_python", "market_quote", ...]
    total_tool_calls INT,
    successful_tool_calls INT,
    failed_tool_calls INT,
    total_cost DECIMAL(10, 4),
    created_at TIMESTAMP DEFAULT NOW(),

    FOREIGN KEY (session_id) REFERENCES sessions(id),
    FOREIGN KEY (user_id) REFERENCES users(id),
    INDEX idx_session_id (session_id),
    INDEX idx_user_id (user_id)
);

-- 10. GATEWAY_PREFERENCES
CREATE TABLE gateway_preferences (
    id UUID PRIMARY KEY,
    user_id UUID NOT NULL UNIQUE REFERENCES users(id),
    model_order JSONB,  -- ["openai", "anthropic", "google"]
    preferred_strategy VARCHAR(50),  -- reasoning, speed, cost
    language VARCHAR(10) DEFAULT 'en',
    theme VARCHAR(20) DEFAULT 'auto',
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW(),

    FOREIGN KEY (user_id) REFERENCES users(id),
    INDEX idx_user_id (user_id)
);

-- 11. MARKET_HISTORY
CREATE TABLE market_history (
    id UUID PRIMARY KEY,
    symbol VARCHAR(20) NOT NULL,
    quote_time TIMESTAMP NOT NULL,
    price DECIMAL(12, 2),
    bid_price DECIMAL(12, 2),
    ask_price DECIMAL(12, 2),
    volume BIGINT,
    change_percent DECIMAL(5, 2),
    created_at TIMESTAMP DEFAULT NOW(),

    INDEX idx_symbol_time (symbol, quote_time DESC),
    INDEX idx_created_at (created_at)
);

-- 12. PROVIDER_FILE_HANDLES
CREATE TABLE provider_file_handles (
    id UUID PRIMARY KEY,
    file_id UUID NOT NULL UNIQUE REFERENCES files(id),
    provider VARCHAR(50),  -- gcs, s3, minio
    provider_path VARCHAR(500),  -- Cloud storage path
    bucket_name VARCHAR(255),
    created_at TIMESTAMP DEFAULT NOW(),
```

```
    FOREIGN KEY (file_id) REFERENCES files(id),
    INDEX idx_provider (provider)
);
```

SECURITY & AUTH SYSTEM

JWT Authentication

```
class AuthService:
    """Authentication with JWT tokens."""

    async def login(self, email: str, password: str) -> dict:
        """Login user."""

        # 1. Find user
        user = await User.query.filter(User.email == email).first()
        if not user:
            raise InvalidCredentials()

        # 2. Verify password
        if not await self.verify_password(password, user.password_hash):
            raise InvalidCredentials()

        # 3. Generate JWT token
        token = await self.create_jwt_token(
            user_id=user.id,
            email=user.email,
            permissions=user.permissions
        )

        # 4. Track login
        user.last_login = datetime.now()
        await db.commit()

        return {
            "token": token,
            "user": user.to_dict(),
            "expires_in": 604800  # 7 days
        }

    def create_jwt_token(self, user_id: UUID, email: str, permissions: List[str]) -> str:
        """Create JWT token."""

        payload = {
            "sub": str(user_id),
            "email": email,
            "permissions": permissions,
            "exp": datetime.utcnow() + timedelta(days=7),
            "iat": datetime.utcnow()
        }

        token = jwt.encode(
            payload,
            settings.JWT_SECRET_KEY,
            algorithm=settings.JWT_ALGORITHM
        )

        return token
```

3-Tier Rate Limiting

```
class RateLimiter:
    """Rate limiting at 3 levels."""

    LIMITS = {
        "per_user_per_minute": 100,
        "per_ip_per_minute": 30,
        "global_per_minute": 10000,
    }

    async def check_limit(self, request: Request, user: Optional[User]) -> bool:
        """Check all rate limit tiers."""

        # Tier 1: Per user (authenticated)
        if user:
            key = f"ratelimit:user:{user.id}"
            count = await redis.incr(key)
            if count == 1:
                await redis.expire(key, 60)

            if count > self.LIMITS["per_user_per_minute"]:
                raise RateLimitExceeded("User limit exceeded")

        # Tier 2: Per IP (all users)
        key = f"ratelimit:ip:{request.client.host}"
        count = await redis.incr(key)
        if count == 1:
            await redis.expire(key, 60)

        if count > self.LIMITS["per_ip_per_minute"]:
            raise RateLimitExceeded("IP limit exceeded")

        # Tier 3: Global
        key = "ratelimit:global"
        count = await redis.incr(key)
        if count == 1:
            await redis.expire(key, 60)

        if count > self.LIMITS["global_per_minute"]:
            raise RateLimitExceeded("System limit exceeded")

        return True
```

Permission Model

```
class PermissionModel:
    """Three-tier permission: ALLOW, DENY, ASK."""

    async def check(
        self,
        user: User,
        tool: str,
        parameters: dict
    ) -> PermissionType:
        """Check if tool execution is allowed."""

        # 1. User role permissions
        if tool not in user.permissions:
            return PermissionType.DENY

        # 2. Parameter safety check
        if tool == "execute_python":
            dangerous_imports = ["os.system", "subprocess", "eval", "exec"]
            code = parameters.get("code", "")

            for dangerous in dangerous_imports:
                if dangerous in code:
                    return PermissionType.ASK  # Ask user first

        # 3. Rate limits
        rate_limited = await self.check_rate_limit(user.id, tool)
        if rate_limited:
            return PermissionType.DENY

        return PermissionType.ALLOW
```

REAL-TIME COMMUNICATION

Server-Sent Events (SSE) Streaming

```python
@router.post("/api/v1/chat/sessions/{session_id}/messages")
async def send_message(session_id: UUID, request: MessageRequest):
    """Send message and stream agent response."""

    async def event_generator():
        try:
            # 1. Save user message
            message = await chat_service.add_message(
                session_id=session_id,
                role="user",
                content=request.content,
                attachments=request.attachments
            )

            yield f"event: message_saved\ndata: {message.id}\n\n"

            # 2. Stream agent response
            async for chunk in agent_executor.execute(
                messages=await chat_service.get_messages(session_id),
                user_id=current_user.id,
                session_id=session_id
            ):
                yield chunk

        except Exception as e:
            yield f"event: error\ndata: {str(e)}\n\n"

    return StreamingResponse(
        event_generator(),
        media_type="text/event-stream",
        headers={
            "Cache-Control": "no-cache",
            "Connection": "keep-alive",
            "X-Accel-Buffering": "no"  # Disable proxy buffering
        }
    )
```

WebSocket (Future)

```python
@router.websocket("/ws/chat/{session_id}")
async def websocket_endpoint(websocket: WebSocket, session_id: UUID):
    """Real-time bidirectional chat."""

    await websocket.accept()

    try:
        while True:
            # Receive message
            data = await websocket.receive_json()

            # Stream agent response
            async for chunk in agent_executor.execute(...):
                await websocket.send_json({
                    "type": "agent_token",
                    "data": chunk
                })

    except WebSocketDisconnect:
        print(f"Client disconnected")
```

DEPLOYMENT & INFRASTRUCTURE

12 Services Docker Compose

```
version: '3.9'

services:
  # Data Layer
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: lumina
      POSTGRES_USER: lumina_user
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "lumina_user"]
      interval: 10s
      timeout: 5s
      retries: 5

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s

  qdrant:
    image: qdrant/qdrant:latest
    ports:
      - "6333:6333"
    volumes:
      - qdrant_data:/qdrant/storage
    environment:
      QDRANT_API_KEY: ${QDRANT_API_KEY}

  minio:
    image: minio/minio:latest
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: ${MINIO_PASSWORD}
    ports:
      - "9000:9000"
      - "9001:9001"
    volumes:
      - minio_data:/minio/data
    command: server /minio/data --console-address ":9001"

  # API Layer
  api:
    build: .
    environment:
      DATABASE_URL: postgresql://lumina_user:${DB_PASSWORD}@db:5432/lumina
      REDIS_URL: redis://redis:6379
      QDRANT_URL: http://qdrant:6333
      PRIMARY_LLM_API_KEY: ${PRIMARY_LLM_API_KEY}
    ports:
      - "8000:8000"
    depends_on:
      db:
```

```
      condition: service_healthy
    redis:
      condition: service_healthy
  command: gunicorn -w 4 -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:8000 app.main:app

worker:
  build: .
  environment:
    DATABASE_URL: postgresql://lumina_user:${DB_PASSWORD}@db:5432/lumina
    CELERY_BROKER_URL: amqp://guest:guest@rabbitmq:5672//
    CELERY_RESULT_BACKEND: redis://redis:6379/1
  depends_on:
    - db
    - redis
    - rabbitmq
  command: celery -A app.worker worker -l info

# Message Broker
rabbitmq:
  image: rabbitmq:3.13-alpine
  ports:
    - "5672:5672"
    - "15672:15672"
  environment:
    RABBITMQ_DEFAULT_USER: guest
    RABBITMQ_DEFAULT_PASS: guest
  volumes:
    - rabbitmq_data:/var/lib/rabbitmq
  healthcheck:
    test: ["CMD", "rabbitmq-diagnostics", "-q", "ping"]
    interval: 10s

# Specialized Services
firecrawl:
  image: ghcr.io/firecrawl/firecrawl:latest
  ports:
    - "3002:3002"
  environment:
    LOG_LEVEL: info

tools_web:
  build:
    context: my_project
    dockerfile: Dockerfile
  ports:
    - "5000:5000"
  environment:
    DATABASE_URL: postgresql://lumina_user:${DB_PASSWORD}@db:5432/lumina
  depends_on:
    - db

tools_worker:
  build:
    context: my_project
    dockerfile: Dockerfile.worker
  environment:
    CELERY_BROKER_URL: amqp://guest:guest@rabbitmq:5672//
    DATABASE_URL: postgresql://lumina_user:${DB_PASSWORD}@db:5432/lumina
  depends_on:
    - rabbitmq
    - db
  command: celery -A tasks worker -l info

# Frontend
web:
  build:
```

```
    build:
      context: ./lumina_web
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      NEXT_PUBLIC_API_URL: http://api:8000
    depends_on:
      - api

  # Reverse Proxy
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - ./https_keys:/etc/nginx/certs:ro
    depends_on:
      - api
      - web
```

TESTING & AUTOMATION

35+ Test/Deploy Scripts

```
├─ deploy_oneclick.py/.ps1          # One-click full deployment
├─ verify_agent.py                  # Agent executor tests
├─ verify_streaming.py              # SSE/streaming tests
├─ verify_prod_full.py              # Full production verification
├─ e2e_frontend_backend_smoke.py     # Smoke tests
├─ e2e_advanced_chat_logic.py        # Advanced chat flows
├─ test_sandbox_escape.py            # Security: sandbox isolation
├─ test_rate_limiting.py             # Rate limiting tests
├─ test_sql_injection.py             # SQL injection prevention
├─ test_path_traversal.py            # Path traversal prevention
├─ reset_admin_password.py           # Admin account recovery
├─ backfill_message_attachments.py   # Data migration scripts
├─ cleanup.sh                        # Container cleanup
└─ [15+ more verification/migration scripts]
```

INNOVATIONS & DIFFERENTIATORS

1. Session Branching Tree (Not Linear)

Unlike ChatGPT's linear history, Lumina allows users to branch conversations at any point and explore alternative paths:

```
Main Conversation
├─ Branch A (explore one direction)
│  ├─ Sub-branch A1
│  └─ Sub-branch A2
├─ Branch B (explore another direction)
└─ Branch C (experiment with different approach)
```

2. Multi-Provider LLM Routing

- Automatically tries multiple providers if one fails
- Different providers for different task types (reasoning vs. speed vs. cost)
- User can set preferred order

3. Scientific Computing Integration

- Full MATLAB/Octave support with graphing
- CJK font support for international users
- Automatic fallback to open-source (Octave) if MATLAB unavailable

4. Real-Time Market Data

- Live stock/crypto quotes integrated into conversations
- Economic calendar events
- Flash news alerts
- Integrated directly into agent tools

5. File Deduplication

- SHA256 content hashing prevents duplicate storage
- Saves ~50% storage costs
- Instant "re-uploads" of identical files

6. Comprehensive Export

- Export conversations to DOCX, PDF, TXT, Markdown
- Preserve formatting, code blocks, thinking process
- Multi-format support not available in ChatGPT

7. Learning Platform

- Auto-generate quizzes from content
- Media compression with tiered quotas
- Web scraping with Firecrawl
- Research tasks with async processing

8. Permission Model

- Three-tier permissions (ALLOW, DENY, ASK)
- Not binary like other systems
- Allows "dangerous but approved" operations

9. Sandbox Isolation

- Per-session sandboxes prevent file access between users
- Complete resource isolation (CPU, memory, timeout)
- Multi-language execution (Python, JavaScript, MATLAB, Bash)

10. Cost Transparency

- Pre-authorize credits before execution
- Settle actual cost post-execution
- Per-message cost breakdown
- Provider-specific pricing

COMPARISON WITH SIMILAR PROJECTS

vs. ChatGPT

Feature	ChatGPT	Lumina
Conversation branching	✗ Linear only	✓ Full tree structure
Code execution	⚠ Limited	✓ Full Python/JS/MATLAB/Bash
Scientific computing	✗ No MATLAB	✓ Full MATLAB/Octave + graphing
Market data	✗ No	✓ Real-time quotes + calendar
Export formats	✗ No	✓ DOCX/PDF/TXT/Markdown
Multi-provider routing	✗ Single provider	✓ 4+ providers with fallback

Feature	ChatGPT	Lumina
Learning platform	✗ No	✓ Quiz generation, compression
Self-hosted option	✗ Cloud only	✓ Full Docker deployment
Transparent pricing	✗ Black box	✓ Per-message cost breakdown

vs. Claude

Feature	Claude	Lumina
Extended thinking	✓ Yes	✓ Yes (via routing)
Code execution	✗ Limited	✓ Full sandbox execution
Session branching	✗ No	✓ Full tree structure
Scientific computing	✗ No	✓ MATLAB/Octave
Self-hosted	✗ Cloud only	✓ Complete self-hosted
Async job processing	✗ No	✓ Celery + RabbitMQ
File versioning	✗ No	✓ Full version control
Web scraping	✗ No	✓ Self-hosted Firecrawl

vs. n8n (Workflow Platform)

Feature	n8n	Lumina
Workflow-first	✓ Yes	✗ Conversation-first
Visual workflow builder	✓ Yes	✗ Natural language
300+ integrations	✓ Yes	⚠ Focused integrations
Agent reasoning	✗ No	✓ Full ReAct loop
Multi-provider LLM	✗ No	✓ Built-in routing
Self-hosted	✓ Yes	✓ Yes
Cost efficiency	⚠ Complex setup	✓ Simpler deployment

vs. LangChain

Feature	LangChain	Lumina
Framework	✓ Flexible framework	✗ Not a framework
Production-ready	⚠ Build required	✓ Ready to deploy
UI included	✗ Build required	✓ Full Next.js UI
Database models	✗ Generic	✓ Domain-specific
Multi-provider routing	✓ Possible	✓ Built-in + optimized
		✓ Optimized ReAct

Feature	LangChain	Lumina
Agent system	✓ Possible	loop
Learning curve	⚠ Steep	✓ Gentle

TECH STACK SUMMARY

```
Backend Layer:
├─ FastAPI (Python 3.11)
├─ SQLAlchemy 2.0 ORM
├─ Pydantic (validation)
├─ asyncio (async runtime)
└─ httpx (async HTTP client)

Database Layer:
├─ PostgreSQL 15 (primary)
├─ Redis 7 (cache + queues)
├─ Qdrant (vector embeddings)
└─ MinIO (object storage)

Execution Layer:
├─ Python executor (subprocess sandbox)
├─ Node.js executor
├─ MATLAB/Octave executor
└─ Bash executor

Feature Layers:
├─ Agent executor (ReAct loop)
├─ LLM gateway (multi-provider routing)
├─ MATLAB integration
├─ Market data service
├─ Research service (Firecrawl)
├─ Media compression (FFmpeg)
├─ Export service (DOCX/PDF)
└─ File versioning

Async Processing:
├─ Celery 5.x
├─ RabbitMQ 3.13
└─ Redis 7

Frontend Layer:
├─ Next.js 14 (App Router)
├─ React 18
├─ TypeScript 5
├─ Tailwind CSS 3.4
├─ Zustand (state management)
└─ TanStack Query (server state)

Infrastructure:
├─ Docker + Docker Compose
├─ Nginx (reverse proxy)
├─ Gunicorn (WSGI server)
├─ Uvicorn (ASGI worker)
├─ Alembic (migrations)
└─ pytest + Playwright (testing)

Security:
├─ JWT authentication
├─ Bcrypt hashing
├─ Rate limiting (3-tier)
├─ Sandbox isolation
├─ Turnstile CAPTCHA
└─ CORS hardening
```

```
    CORS hardening

External Services:
├─ OpenAI API
├─ Anthropic API
├─ Google Gemini API
├─ DeepSeek API
├─ Serper API (search)
├─ Perplexity API (search fallback)
├─ Groq Whisper (transcription)
├─ Google Cloud Storage
└─ Firecrawl (web scraping)
```

CONCLUSION

Luminai.chat is a **comprehensive AI-powered workspace** that goes far beyond a simple chat system. It combines:

1. **Intelligent conversation** with multi-provider LLM routing
2. **Code execution** in secure, isolated sandboxes
3. **Scientific computing** with full MATLAB/Octave support
4. **Financial data** integration with real-time market quotes
5. **Research tools** with web scraping and academic search
6. **Document management** with version control and multi-format export
7. **Learning platform** with quiz generation and media processing
8. **Enterprise features** with role-based access, billing, and team collaboration

The project demonstrates:

- **Production-grade architecture** with 12 microservices
- **Security best practices** with sandboxing and permission models
- **Scalability design** with horizontal scaling and async processing
- **User-centric innovation** with session branching and multi-provider routing
- **Complete DevOps** with one-click deployment and 35+ automation scripts

Total Codebase: 45,000+ LOC across backend, frontend, research services, and infrastructure.

Last Updated: 2026-04-20