

# Agentic AI Architecture

Autonomous Marketing Campaign Management

*"Marketing on autopilot — AI agents that think, create, optimize, and scale"*

# 1. Agentic AI Vision

## What is Agentic AI?

Agentic AI represents autonomous AI systems that can perceive, reason, plan, and act independently to achieve complex goals. Unlike traditional AI that responds to prompts, agentic systems proactively manage workflows, make decisions, and continuously optimize without constant human intervention.

## Automation Target

Current marketing platforms achieve 40-60% automation. MarketingAI targets **85%+ automation** through specialized AI agents.



Task Category	Current Manual	With Agentic AI	Automation %
Content Ideation	2-3 hours/week	10 mins review	95%
Content Creation	10-15 hours/week	1-2 hours editing	90%
Visual Design	5-8 hours/week	30 mins approval	92%
Scheduling	2-3 hours/week	Fully automated	100%
Performance Analysis	3-5 hours/week	Auto-generated reports	95%
Optimization	4-6 hours/week	Continuous AI optimization	90%
Engagement Response	5-10 hours/week	AI drafts, human approval	70%
TOTAL	31-50 hrs/week	4-6 hrs/week	85-90%

# 2. Multi-Agent System Architecture





3. Specialized Agent Definitions



Strategy Agent

CORE

**Purpose:** High-level campaign planning, market research, and strategic decision-making.

Capability	Implementation	Automation Level
Campaign Planning	Gemini 2.5 Pro with extended thinking	95%
Competitor Analysis	Web scraping + NLP analysis	90%
Trend Detection	Social listening + prediction models	85%
Audience Research	Analytics synthesis + segmentation	90%
Content Calendar	Auto-generated based on goals	95%

Agent Workflow:



**Goal Analysis:** Parse user's business goals and constraints

2 **Market Research:** Analyze competitors, trends, audience behavior

3 **Strategy Generation:** Create multi-week content strategy

4 **Calendar Population:** Schedule content themes and types

5 **Delegation:** Assign tasks to Content and Visual agents

```
# Strategy Agent Implementation
class StrategyAgent:
    def __init__(self, brand_context, market_data):
        self.model = GenerativeModel("gemini-2.5-pro-preview")
        self.brand = brand_context
        self.market = market_data
        self.memory = AgentMemory()

    async def generate_campaign_strategy(self, goals, duration_weeks=4):
        """Generate comprehensive campaign strategy"""
        # Gather context
        competitor_analysis = await self.analyze_competitors()
        trend_data = await self.fetch_trends()
        audience_insights = await self.analyze_audience()

        # Generate strategy with extended thinking
        strategy_prompt = f"""
        Brand: {self.brand.name}
        Industry: {self.brand.industry}
        Goals: {goals}
        Duration: {duration_weeks} weeks
        Competitor Insights: {competitor_analysis}
        Current Trends: {trend_data}
        Audience: {audience_insights}
        Generate a detailed content marketing strategy including:
        1. Weekly themes and focus areas
        2. Content mix (posts, reels, stories) per platform
        3. Key messages and CTAs
        4. Optimal posting schedule
        5. KPIs and success metrics
        """
        response = await self.model.generate_content_async(
            strategy_prompt, generation_config={"temperature": 0.7}
        )
        return self.parse_strategy(response.text)

    async def delegate_to_content_agent(self, strategy):
        """Break down strategy into content tasks"""
        tasks = []
        for week in strategy.weeks:
            for content_item in week.content_items:
                tasks.append(
                    {
                        "type": content_item.type,
                        "platform": content_item.platform,
                        "theme": content_item.theme,
                        "deadline": content_item.scheduled_date,
                        "priority": content_item.priority
                    }
                )
        return tasks
```



**Content Agent** CORE

**Purpose:** Generate, adapt, and optimize written content across all platforms.

Capability	Implementation	Automation Level
Caption Generation	Gemini Flash for speed	95%
Blog Writing	Gemini Pro for quality	85%
Platform Adaptation	Auto-resize and reformat	100%
Hashtag Optimization	Trend analysis + performance data	95%
A/B Variants	Auto-generate multiple versions	100%
Brand Voice Match	Fine-tuned on brand content	90%

```
# Content Agent Implementation
class ContentAgent:
    def __init__(self, brand_voice_model):
        self.fast_model = GenerativeModel("gemini-2.0-flash")
        self.quality_model = GenerativeModel("gemini-2.5-pro-preview")
        self.brand_voice = brand_voice_model

    async def generate_social_post(self, brief, platform, variations=3):
        """Generate platform-optimized social media posts"""
        platform_specs = PLATFORM_SPECS[platform]
        prompt = f"""
        Brand Voice: {self.brand_voice.description}
        Platform: {platform}
        Character Limit: {platform_specs['char_limit']}
        Best Practices: {platform_specs['best_practices']}
        Brief: {brief}
        Generate {variations} unique post variations that:
        1. Match the brand voice perfectly
        2. Optimize for {platform} algorithm
        3. Include
        """
```

```
relevant hashtags 4. Have compelling CTAs 5. Are different enough for A/B testing """ response = await
self.fast_model.generate_content_async(prompt) posts = self.parse_variations(response.text) # Score each variation for post in
posts: post.predicted_engagement = await self.predict_engagement(post) return sorted(posts, key=lambda x: x.predicted_engagement,
reverse=True) async def adapt_content_to_platform(self, original_content, target_platform): """Automatically adapt content for
different platforms""" adaptations = {} for platform in target_platform: adapted = await
self.fast_model.generate_content_async(f""" Original: {original_content} Adapt for: {platform} Requirements:
{PLATFORM_SPECS[platform]} Maintain: Core message, brand voice Optimize: Format, length, hashtags, CTAs """) adaptations[platform]
= adapted.text return adaptations
```



Visual Agent

CORE

**Purpose:** Generate and edit images, videos, and graphics for all content.

Capability	Implementation	Automation Level
Image Generation	Imagen 3 + Canva API	90%
Video Creation	Veo 2 for short clips	85%
Template Application	Canva Autofill API	95%
Brand Consistency	Brand kit enforcement	100%
Format Adaptation	Auto-resize for platforms	100%
Thumbnail Generation	AI composition + text overlay	90%

```
# Visual Agent Implementation class VisualAgent: def __init__(self, brand_kit): self.imagen =
ImageGenerationModel.from_pretrained("imagen-3.0-generate-002") self.veo = VideoGenerationModel.from_pretrained("veo-2.0-generate-
001") self.canva = CanvaConnectAPI() self.brand_kit = brand_kit async def generate_post_visual(self, content, platform,
style="brand"): """Generate visual content based on post text""" # Analyze content for visual elements visual_brief = await
self.analyze_visual_needs(content) # Generate with Imagen 3 prompt = f""" Style: {self.brand_kit.visual_style} Colors:
{self.brand_kit.colors} Content Theme: {visual_brief.theme} Elements: {visual_brief.suggested_elements} Mood: {visual_brief.mood}
Aspect Ratio: {PLATFORM_SPECS[platform]['aspect_ratio']} """ images = self.imagen.generate_images( prompt=prompt,
number_of_images=4, aspect_ratio=PLATFORM_SPECS[platform]['aspect_ratio'] ) # Rank by brand alignment scored_images = await
self.score_brand_alignment(images) return scored_images[0] # Return best match async def create_video_reel(self, script,
duration=15): """Generate short-form video content""" video = self.veo.generate_video( prompt=f""" Script: {script} Brand Style:
{self.brand_kit.video_style} Duration: {duration} seconds Format: Vertical 9:16 for Reels/TikTok Pacing: Fast, engaging, hook in
first 3 seconds """, duration_seconds=duration, aspect_ratio="9:16" ) # Add branded elements via Canva final_video = await
self.canva.add_brand_overlay( video, logo=self.brand_kit.logo, colors=self.brand_kit.colors ) return final_video
```



Publishing Agent

AUTO

**Purpose:** Automated scheduling, posting, and engagement management.

Capability	Implementation	Automation Level
Smart Scheduling	ML-based optimal time prediction	100%

Multi-Platform Posting	Social media API integrations	100%
Comment Monitoring	Real-time webhook processing	100%
Reply Drafting	Context-aware AI responses	80%
Crisis Detection	Sentiment monitoring + alerts	95%
Engagement Actions	Auto-like, follow patterns	90%

```
# Publishing Agent Implementation
class PublishingAgent:
    def __init__(self, social_accounts):
        self.platforms = {
            'instagram': InstagramGraphAPI(social_accounts['instagram']),
            'facebook': MetaGraphAPI(social_accounts['facebook']),
            'twitter': TwitterAPIv2(social_accounts['twitter']),
            'linkedin': LinkedInAPI(social_accounts['linkedin']),
            'tiktok': TikTokAPI(social_accounts['tiktok']),
            'youtube': YouTubeDataAPI(social_accounts['youtube'])
        }
        self.scheduler = SmartScheduler()
        self.engagement_bot = EngagementManager()

    async def schedule_content(self, content, platforms):
        """Schedule content at optimal times"""
        scheduled = []
        for platform in platforms:
            # Get AI-predicted optimal time
            optimal_time = await self.scheduler.predict_best_time(
                platform=platform,
                content_type=content.type,
                audience=content.target_audience,
                historical_data=await self.get_performance_history(platform)
            )
            # Adapt content for platform
            platform_content = await content.adapt_for(platform)
            # Schedule via platform API
            post_id = await self.platforms[platform].schedule_post(
                content=platform_content,
                scheduled_time=optimal_time
            )
            scheduled.append({
                'platform': platform,
                'post_id': post_id,
                'scheduled_time': optimal_time
            })
        return scheduled

    async def monitor_and_respond(self):
        """Continuous engagement monitoring and response"""
        while True:
            for platform, api in self.platforms.items():
                # Check for new comments/mentions notifications
                notifications = await api.get_notifications()
                for notif in notifications:
                    # Analyze sentiment
                    sentiment = await self.analyze_sentiment(notif.content)
                    if sentiment.is_negative and sentiment.score < -0.7:
                        # Alert human for crisis potential
                        await self.alert_human(notif, priority="high")
                    else:
                        # Generate AI response
                        response = await self.generate_response(notif)
                        if sentiment.is_positive:
                            # Auto-post positive responses
                            await api.reply(notif.id, response)
                        else:
                            # Queue for human review
                            await self.queue_for_review(notif, response)
            await asyncio.sleep(60) # Check every minute
```



Analytics Agent

AI

**Purpose:** Real-time performance tracking, reporting, and insight generation.

Capability	Implementation	Automation Level
Data Collection	Platform API aggregation	100%
Report Generation	Automated daily/weekly/monthly	100%
Insight Extraction	Gemini analysis of patterns	95%
Anomaly Detection	ML-based spike/drop alerts	95%
ROI Calculation	Attribution modeling	90%
Recommendations	AI-generated action items	85%

```
# Analytics Agent Implementation
class AnalyticsAgent:
    def __init__(self):
        self.data_collector = MultiPlatformDataCollector()
        self.analyzer = GenerativeModel("gemini-2.5-pro-preview")
        self.anomaly_detector = AnomalyDetectionModel()

    async def generate_performance_report(self, period="weekly"):
        """Generate comprehensive performance report"""
        # Collect data from all platforms
        raw_data = await self.data_collector.fetch_all(period)
        # Calculate metrics
        metrics = {
            'total_reach': sum(d.reach for d in raw_data),
            'total_engagement': sum(d.engagement for d in raw_data),
            'engagement_rate': self.calculate_engagement_rate(raw_data),
            'top_performing': self.rank_content(raw_data)[:5],
            'worst_performing': self.rank_content(raw_data)[-5:],
            'platform_breakdown': self.breakdown_by_platform(raw_data),
            'content_type_analysis': self.analyze_by_type(raw_data)
        }
        # Generate AI insights
        insights = await self.analyzer.generate_content_async(f"Analyze this
```

```
marketing performance data and provide: 1. Key wins and successes 2. Areas needing improvement 3. Specific actionable recommendations 4. Predicted trends for next period 5. Budget reallocation suggestions Data: {json.dumps(metrics)} """ return Report(metrics=metrics, insights=insights.text) async def detect_anomalies(self): """Real-time anomaly detection""" current_metrics = await self.data_collector.fetch_realtime() anomalies = self.anomaly_detector.detect(current_metrics) for anomaly in anomalies: if anomaly.type == "positive_spike": # Viral content - double down await self.notify_optimization_agent( action="boost", content_id=anomaly.content_id ) elif anomaly.type == "negative_spike": # Crisis potential await self.alert_human(anomaly, priority="urgent")
```



Optimization Agent

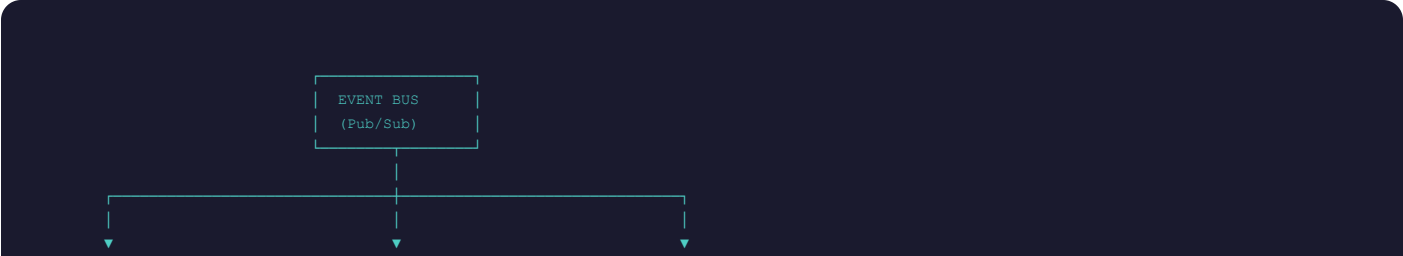
AUTO

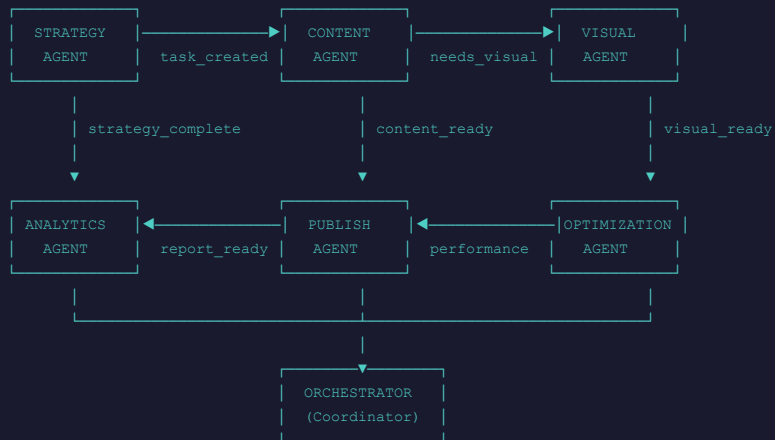
**Purpose:** Continuous optimization of content, timing, targeting, and budget.

Capability	Implementation	Automation Level
A/B Test Management	Automated variant testing	100%
Performance Prediction	ML engagement forecasting	90%
Budget Optimization	Dynamic allocation algorithms	85%
Timing Refinement	Continuous schedule optimization	95%
Audience Targeting	Lookalike and interest expansion	85%
Content Ranking	Quality score calculation	95%

```
# Optimization Agent Implementation class OptimizationAgent: def __init__(self): self.ab_test_manager = ABTestManager() self.predictor = EngagementPredictor() self.budget_optimizer = BudgetAllocator() async def run_continuous_optimization(self): """Main optimization loop""" while True: # 1. Check A/B test results active_tests = await self.ab_test_manager.get_active_tests() for test in active_tests: if test.has_statistical_significance(): winner = test.get_winner() await self.promote_winner(winner) await self.archive_losers(test) # 2. Optimize posting times performance_data = await self.get_recent_performance() new_schedule = self.recalculate_optimal_times(performance_data) await self.update_schedule(new_schedule) # 3. Reallocate budget roi_by_platform = await self.calculate_platform_roi() new_allocation = self.budget_optimizer.optimize(roi_by_platform) await self.apply_budget_changes(new_allocation) # 4. Content quality scoring pending_content = await self.get_pending_content() for content in pending_content: score = await self.predictor.predict_engagement(content) if score < MINIMUM_QUALITY_THRESHOLD: await self.flag_for_revision(content) else: await self.approve_for_publishing(content) await asyncio.sleep(3600) # Run hourly async def auto_ab_test(self, content): """Automatically create and manage A/B tests""" # Generate variants variants = await ContentAgent().generate_variants(content, count=3) # Create test test = await self.ab_test_manager.create_test( variants=variants, metric="engagement_rate", traffic_split=[0.33, 0.33, 0.34], min_sample_size=1000, max_duration_hours=48 ) return test.id
```

4. Inter-Agent Communication





## EVENT TYPES:

- strategy\_complete → Triggers content creation tasks
- task\_created → Assigns work to content agent
- content\_ready → Triggers visual generation
- needs\_visual → Requests visual agent work
- visual\_ready → Marks content complete
- content\_complete → Triggers scheduling
- post\_published → Starts performance tracking
- performance\_update → Triggers optimization
- anomaly\_detected → Alerts orchestrator
- optimization\_action → Applies changes

## 5. Memory & Learning System

### Continuous Learning Architecture

The Learning Agent maintains institutional memory across all agents, enabling the system to improve over time based on performance data.

```
# Shared Memory & Learning System class AgentMemorySystem: def __init__(self): self.short_term = RedisCache() # Recent context (24 hours) self.long_term = BigQueryStore() # Historical data self.knowledge_base = VertexAISearch() # RAG system self.embeddings = TextEmbeddingModel() async def store_success_pattern(self, content, performance): """Learn from successful content""" if performance.engagement_rate > THRESHOLD_HIGH: pattern = await self.extract_pattern(content) # Store in knowledge base await self.knowledge_base.add_document( content=content.to_dict(), performance=performance.to_dict(), pattern=pattern, embedding=await self.embeddings.encode(content.text) ) # Update brand voice model await self.update_brand_model(content, "positive") async def retrieve_similar_successes(self, new_content_brief): """Find similar past successes for inspiration""" query_embedding = await self.embeddings.encode(new_content_brief) similar = await self.knowledge_base.search( embedding=query_embedding, filters= {"engagement_rate": {">": THRESHOLD_HIGH}}, limit=5 ) return similar async def get_brand_patterns(self, brand_id): """Retrieve learned patterns for a brand""" return await self.long_term.query(f""" SELECT content_type, avg_engagement, best_posting_times, top_hashtags, winning_cta_patterns, visual_style_scores FROM learned_patterns WHERE brand_id = '{brand_id}' ORDER BY last_updated DESC LIMIT 1 """)
```

## 6. Human-in-the-Loop Controls

Control Point	Default Setting	Override Options	Recommendation
Content Approval	Auto-approve high-score	All manual, score threshold	Threshold: 80+
Visual Approval	Auto-approve on-brand	All manual, brand-score	Threshold: 85+

Engagement Replies	Queue all for review	Auto-positive, manual-negative	Auto positive only
Budget Changes	Alert on >10% change	All manual, auto-optimize	Alert >20%
Crisis Detection	Always alert human	N/A	Never auto-handle
New Campaign Launch	Require approval	Auto-launch scheduled	Approval required

7. Implementation Phases

Phase	Agents	Duration	Automation %
Phase 1: Foundation	Content Agent, Visual Agent	4-5 weeks	60%
Phase 2: Publishing	Publishing Agent, Scheduler	3-4 weeks	70%
Phase 3: Analytics	Analytics Agent, Reporting	3-4 weeks	75%
Phase 4: Strategy	Strategy Agent, Planning	4-5 weeks	80%
Phase 5: Optimization	Optimization Agent, A/B Testing	4-5 weeks	85%
Phase 6: Learning	Learning Agent, Memory System	4-5 weeks	85-90%
TOTAL	Full Agentic System	22-28 weeks	85-90%