🤖 GitHub Copilot Pro Instructions for Gemini-Flow

This document provides comprehensive instructions for GitHub Copilot Pro to effectively contribute to the Gemini-Flow repository using Model Context Protocol (MCP) server integrations.

📋 Project Context

Gemini-Flow is a revolutionary multi-model AI orchestration platform that enables:
- **High-Performance Orchestration**: SQLite-based architecture with 396K ops/sec
- **Quantum-Classical Hybrid Processing**: Integration of quantum and classical computing paradigms
- **Distributed AI Swarms**: Coordinated multi-agent systems with Byzantine consensus
- **MCP Integration**: Seamless integration with external services through Model Context Protocol

Repository Structure
```
gemini-flow/
├── src/
│   ├── cli/                # CLI interface and commands
│   ├── services/           # Core services and integrations
│   ├── core/               # MCP adapters and core functionality
│   ├── agents/             # Agent implementations
│   └── types/              # TypeScript type definitions
├── docs/                   # Documentation and architecture
├── scripts/                # Build and utility scripts
├── tests/                  # Test suites
└── .hive-mind/            # SQLite database and session storage
```

🔌 MCP Server Integrations

GitHub Copilot Pro should leverage these 7 MCP servers for enhanced functionality:

1. Redis MCP Server (Caching & State Management)
```json
{
 "redis": {
   "command": "node",
   "args": ["/path/to/redis-mcp/index.js"],
   "env": {
```

```
      "REDIS_HOST": "localhost",
      "REDIS_PORT": "6379"
    }
  }
}
```

Usage Patterns:
```typescript
// Cache frequently accessed data
async function cacheSwarmState(swarmId: string, state: any) {
  return await mcp.redis.set(`swarm:${swarmId}`, JSON.stringify(state), {
    EX: 3600 // 1 hour TTL
  });
}

// Implement distributed locks
async function acquireSwarmLock(swarmId: string): Promise<boolean> {
  const lockKey = `lock:swarm:${swarmId}`;
  const lockId = crypto.randomUUID();
  const acquired = await mcp.redis.set(lockKey, lockId, {
    NX: true,  // Only set if not exists
    EX: 30     // 30 second lock timeout
  });
  return acquired === 'OK';
}

// Performance optimization: Pipeline multiple operations
async function bulkCacheAgentStates(agents: Agent[]) {
  const pipeline = mcp.redis.pipeline();
  agents.forEach(agent => {
    pipeline.hset(`agent:${agent.id}`, {
      status: agent.status,
      lastActive: Date.now(),
      metrics: JSON.stringify(agent.metrics)
    });
  });
  return await pipeline.exec();
}
```

2. Puppeteer MCP Server (Browser Automation & Testing)

```json
{
 "puppeteer": {
   "command": "node",
   "args": ["/path/to/puppeteer-mcp/index.js"],
   "env": {
     "PUPPETEER_HEADLESS": "true",
     "PUPPETEER_TIMEOUT": "30000"
   }
 }
}
```

Usage Patterns:
```typescript
// Automated UI testing for web interfaces
async function testQuantumVisualization() {
 const browser = await mcp.puppeteer.launch({ headless: true });
 const page = await browser.newPage();
  await page.goto('http://localhost:3000/quantum-viz');
  // Test quantum circuit rendering
 await page.waitForSelector('.quantum-circuit-canvas');
 const circuitRendered = await page.evaluate(() => {
   const canvas = document.querySelector('.quantum-circuit-canvas');
   return canvas && canvas.children.length > 0;
 });
  assert(circuitRendered, 'Quantum circuit should render');
  // Test real-time updates
 await page.click('#run-quantum-simulation');
 await page.waitForSelector('.simulation-results', { timeout: 10000 });
  await browser.close();
}

// Generate documentation screenshots
async function captureArchitectureDiagrams() {
 const browser = await mcp.puppeteer.launch();
 const page = await browser.newPage();
  await page.goto('http://localhost:3000/architecture');
 await page.setViewport({ width: 1920, height: 1080 });
  // Capture swarm topology visualization
 await page.screenshot({
   path: './docs/images/swarm-topology.png',
```

````
   fullPage: true
 });
  await browser.close();
}
```

### 3. Filesystem MCP Server (File Operations)
```json
{
 "filesystem": {
   "command": "npx",
   "args": ["-y", "@modelcontextprotocol/server-filesystem", "/Users/workspace"]
 }
}
```

**Usage Patterns:**
```typescript
// Analyze project structure
async function analyzeCodebaseStructure() {
 const files = await mcp.filesystem.list('/src');
  const analysis = {
   totalFiles: 0,
   byType: {} as Record<string, number>,
   largeFiles: [] as string[]
 };
  for (const file of files) {
   if (file.isFile) {
     analysis.totalFiles++;
     const ext = path.extname(file.name);
     analysis.byType[ext] = (analysis.byType[ext] || 0) + 1;

     const stats = await mcp.filesystem.stat(file.path);
     if (stats.size > 500 * 1024) { // Files > 500KB
       analysis.largeFiles.push(file.path);
     }
   }
 }
  return analysis;
}

// Batch file operations for agent outputs
````

```
async function saveAgentResults(results: AgentResult[]) {
 const timestamp = new Date().toISOString();
 const outputDir = `/outputs/${timestamp}`;
  await mcp.filesystem.mkdir(outputDir, { recursive: true });
  // Parallel write operations
 await Promise.all(
   results.map(async (result, index) => {
     const filename = `${outputDir}/agent-${result.agentId}-output.json`;
     await mcp.filesystem.write(filename, JSON.stringify(result, null, 2));
   })
 );
  // Create summary file
 const summary = {
   timestamp,
   totalAgents: results.length,
   successCount: results.filter(r => r.success).length,
   outputDirectory: outputDir
 };
  await mcp.filesystem.write(
   `${outputDir}/summary.json`,
   JSON.stringify(summary, null, 2)
 );
}
```

4. GitHub MCP Server (Repository Management)
```json
{
 "github": {
   "command": "npx",
   "args": ["-y", "@modelcontextprotocol/server-github"],
   "env": {
     "GITHUB_PERSONAL_ACCESS_TOKEN": "your-token"
   }
 }
}
```

Usage Patterns:
```typescript
// Automated issue management
async function createSwarmTaskIssue(task: SwarmTask) {
```

```
 const issue = await mcp.github.createIssue({
   owner: 'clduab11',
   repo: 'gemini-flow',
   title: `[Swarm Task] ${task.description}`,
   body: `
## Task Details
- **Task ID**: ${task.id}
- **Priority**: ${task.priority}
- **Assigned Agents**: ${task.agents.join(', ')}
- **Dependencies**: ${task.dependencies.join(', ')}

## Acceptance Criteria
${task.criteria.map(c => `- [ ] ${c}`).join('\n')}

## Automated Tracking
This issue is automatically tracked by swarm ${task.swarmId}
   `,
   labels: ['swarm-task', task.priority, 'automated']
 });
  return issue.number;
}

// PR automation with swarm validation
async function createSwarmValidatedPR(branch: string, changes: FileChange[]) {
 // First, create the branch and commit changes
 await mcp.github.createBranch({
   owner: 'clduab11',
   repo: 'gemini-flow',
   ref: `refs/heads/${branch}`,
   sha: await mcp.github.getDefaultBranchSHA()
 });
  // Commit changes
 for (const change of changes) {
   await mcp.github.createOrUpdateFile({
     owner: 'clduab11',
     repo: 'gemini-flow',
     path: change.path,
     message: change.commitMessage,
     content: Buffer.from(change.content).toString('base64'),
     branch
   });
 }
```

```
  // Create PR with swarm validation checks
 const pr = await mcp.github.createPullRequest({
   owner: 'clduab11',
   repo: 'gemini-flow',
   title: `[Swarm Validated] ${changes[0].commitMessage}`,
   head: branch,
   base: 'main',
   body: `
## Changes
${changes.map(c => `- ${c.path}: ${c.description}`).join('\n')}

## Swarm Validation
- [ ] Code quality check by analyzer agents
- [ ] Security review by security agents
- [ ] Performance validation by benchmark agents
- [ ] Integration tests by test agents

*This PR was created and will be validated by the Gemini-Flow swarm system*
   `
 });
  return pr.number;
}
```

5. Mem0 MCP Server (Memory & Knowledge Graphs)
```json
{
 "mem0": {
   "command": "node",
   "args": ["/path/to/mem0-mcp/index.js"],
   "env": {
     "MEM0_API_KEY": "your-api-key"
   }
 }
}
```

Usage Patterns:
```typescript
// Build agent knowledge graphs
async function buildAgentKnowledgeGraph(agentId: string) {
 const memories = [];
```

```
  // Store agent capabilities
 await mcp.mem0.add({
   agentId,
   memory: {
     type: 'capability',
     content: 'Specialized in quantum circuit optimization',
     metadata: {
       domain: 'quantum-computing',
       proficiency: 0.95
     }
   }
 });
  // Store learned patterns
 await mcp.mem0.add({
   agentId,
   memory: {
     type: 'pattern',
     content: 'Variational Quantum Eigensolver converges faster with COBYLA
optimizer',
     metadata: {
       successRate: 0.87,
       applications: ['molecular-simulation', 'optimization']
     }
   }
 });
  // Create knowledge connections
 await mcp.mem0.connect({
   from: { agentId, type: 'capability' },
   to: { agentId, type: 'pattern' },
   relationship: 'applies-to',
   strength: 0.9
 });
}

// Cross-agent knowledge sharing
async function shareKnowledgeAcrossSwarm(swarmId: string, knowledge: Knowledge) {
 const agents = await getSwarmAgents(swarmId);
  // Store shared knowledge
 const sharedMemory = await mcp.mem0.add({
   swarmId,
   memory: {
     type: 'shared-insight',
```

```
    content: knowledge.content,
    metadata: {
      sourceAgent: knowledge.sourceAgentId,
      timestamp: Date.now(),
      confidence: knowledge.confidence
    }
  }
});
 // Propagate to relevant agents
const relevantAgents = agents.filter(agent =>
  agent.capabilities.some(cap => knowledge.domains.includes(cap))
);
 await Promise.all(
  relevantAgents.map(agent =>
    mcp.mem0.addReference({
      agentId: agent.id,
      memoryId: sharedMemory.id,
      relevance: calculateRelevance(agent, knowledge)
    })
  )
);
}
```

6. Supabase MCP Server (Database Operations)
```json
{
 "supabase": {
   "command": "npx",
   "args": ["-y", "@modelcontextprotocol/server-supabase"],
   "env": {
     "SUPABASE_URL": "your-project-url",
     "SUPABASE_SERVICE_ROLE_KEY": "your-service-key"
   }
 }
}
```

Usage Patterns:
```typescript
// Implement swarm state persistence
async function persistSwarmState(swarm: Swarm) {
```

```
 // Create swarm record
 const { data: swarmRecord, error } = await mcp.supabase
   .from('swarms')
   .upsert({
     id: swarm.id,
     topology: swarm.topology,
     status: swarm.status,
     agent_count: swarm.agents.length,
     created_at: swarm.createdAt,
     updated_at: new Date().toISOString(),
     metadata: swarm.metadata
   });
  if (error) throw error;
  // Batch insert agent states
 const agentRecords = swarm.agents.map(agent => ({
   id: agent.id,
   swarm_id: swarm.id,
   type: agent.type,
   status: agent.status,
   capabilities: agent.capabilities,
   performance_metrics: agent.metrics
 }));
  await mcp.supabase
   .from('agents')
   .upsert(agentRecords);
  // Set up real-time subscriptions
 const subscription = mcp.supabase
   .channel(`swarm-${swarm.id}`)
   .on('postgres_changes', {
     event: '*',
     schema: 'public',
     table: 'agents',
     filter: `swarm_id=eq.${swarm.id}`
   }, (payload) => {
     handleAgentStateChange(payload);
   })
   .subscribe();
  return { swarmRecord, subscription };
}

// Implement Row Level Security for multi-tenant swarms
async function setupSwarmRLS() {
```

```
 // Enable RLS on swarms table
 await mcp.supabase.rpc('enable_rls', {
   table_name: 'swarms'
 });
  // Create policy for swarm access
 const policy = `
   CREATE POLICY "Users can only access their own swarms"
   ON swarms
   FOR ALL
   USING (auth.uid() = user_id OR 'public' = ANY(access_levels));
 `;
  await mcp.supabase.rpc('execute_sql', { query: policy });
}

// High-performance task queue implementation
async function createTaskQueue(swarmId: string) {
 // Create optimized task table with indexes
 await mcp.supabase.rpc('execute_sql', {
   query: `
     CREATE TABLE IF NOT EXISTS task_queue_${swarmId} (
       id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
       task_data JSONB NOT NULL,
       priority INTEGER DEFAULT 0,
       status TEXT DEFAULT 'pending',
       claimed_by UUID,
       claimed_at TIMESTAMPTZ,
       completed_at TIMESTAMPTZ,
       created_at TIMESTAMPTZ DEFAULT NOW(),

       -- Indexes for performance
       INDEX idx_status_priority (status, priority DESC),
       INDEX idx_claimed_by (claimed_by) WHERE claimed_by IS NOT NULL
     );
   `
 });
  // Create function for atomic task claiming
 await mcp.supabase.rpc('execute_sql', {
   query: `
     CREATE OR REPLACE FUNCTION claim_next_task_${swarmId}(agent_id UUID)
     RETURNS JSONB AS $$
     DECLARE
       next_task RECORD;
```

```
    BEGIN
      SELECT * INTO next_task
      FROM task_queue_${swarmId}
      WHERE status = 'pending'
      ORDER BY priority DESC, created_at ASC
      LIMIT 1
      FOR UPDATE SKIP LOCKED;

      IF FOUND THEN
        UPDATE task_queue_${swarmId}
        SET status = 'processing',
            claimed_by = agent_id,
            claimed_at = NOW()
        WHERE id = next_task.id;

        RETURN row_to_json(next_task);
      END IF;

      RETURN NULL;
    END;
    $$ LANGUAGE plpgsql;
  `
 });
}
```

7. MCP-Omnisearch Server (Multi-Provider Search)
```json
{
 "mcp-omnisearch": {
   "command": "npx",
   "args": ["-y", "mcp-omnisearch"],
   "env": {
     "GOOGLE_API_KEY": "your-google-key",
     "ANTHROPIC_API_KEY": "your-anthropic-key"
   }
 }
}
```

Usage Patterns:
```typescript
```

```typescript
// Research quantum computing patterns
async function researchQuantumPatterns(topic: string) {
 const searchResults = await mcp.omnisearch.search({
   query: `${topic} quantum computing optimization techniques`,
   providers: ['google', 'academic', 'github'],
   filters: {
     dateRange: 'past_year',
     fileType: ['pdf', 'ipynb', 'py'],
     domains: ['arxiv.org', 'github.com', 'qiskit.org']
   }
 });
  // Analyze results for patterns
 const patterns = await analyzeSearchResults(searchResults);
  // Store valuable findings in knowledge base
 for (const pattern of patterns) {
   await mcp.mem0.add({
     memory: {
       type: 'research-finding',
       content: pattern.summary,
       source: pattern.source,
       relevance: pattern.relevanceScore
     }
   });
 }
  return patterns;
}

// Multi-source documentation aggregation
async function aggregateDocumentation(component: string) {
 const sources = await mcp.omnisearch.search({
   query: `${component} documentation best practices`,
   providers: ['github', 'stackoverflow', 'documentation'],
   maxResults: 50
 });
  // Process and synthesize documentation
 const synthesized = await synthesizeDocumentation(sources);
  // Generate comprehensive docs
 await mcp.filesystem.write(
   `/docs/generated/${component}-guide.md`,
   synthesized.content
 );
  return synthesized;
```

```
}
```

🎯 Best Practices for Contributing

1. Code Quality Standards
```typescript
// Always use type-safe MCP calls
interface MCPResponse<T> {
 success: boolean;
 data?: T;
 error?: string;
}

async function safeMCPCall<T>(
 operation: () => Promise<T>
): Promise<MCPResponse<T>> {
 try {
   const data = await operation();
   return { success: true, data };
 } catch (error) {
   console.error('MCP operation failed:', error);
   return {
     success: false,
     error: error instanceof Error ? error.message : 'Unknown error'
   };
 }
}

// Example usage
const result = await safeMCPCall(() =>
 mcp.redis.get('swarm:state')
);

if (result.success) {
 processSwarmState(result.data);
}
```

2. Performance Optimization
```typescript
// Implement connection pooling
```

```
class MCPConnectionPool {
 private connections: Map<string, any> = new Map();
 private maxConnections = 10;
  async getConnection(server: string) {
   if (!this.connections.has(server)) {
     const conn = await this.createConnection(server);
     this.connections.set(server, conn);
   }
   return this.connections.get(server);
 }
  private async createConnection(server: string) {
   // Implement connection logic based on server type
   switch (server) {
     case 'redis':
       return await mcp.redis.connect();
     case 'supabase':
       return await mcp.supabase.initialize();
     // ... other servers
   }
 }
}

// Batch operations for efficiency
async function batchProcessTasks(tasks: Task[]) {
 const BATCH_SIZE = 100;
 const batches = [];
  for (let i = 0; i < tasks.length; i += BATCH_SIZE) {
   batches.push(tasks.slice(i, i + BATCH_SIZE));
 }
  const results = await Promise.all(
   batches.map(batch => processBatch(batch))
 );
  return results.flat();
}
```

3. Error Handling and Resilience
```typescript
// Implement circuit breaker pattern
class CircuitBreaker {
 private failures = 0;
 private lastFailTime = 0;
```

```
 private state: 'closed' | 'open' | 'half-open' = 'closed';
  async execute<T>(operation: () => Promise<T>): Promise<T> {
   if (this.state === 'open') {
     if (Date.now() - this.lastFailTime > 60000) { // 1 minute
       this.state = 'half-open';
     } else {
       throw new Error('Circuit breaker is open');
     }
   }

   try {
     const result = await operation();
     if (this.state === 'half-open') {
       this.state = 'closed';
       this.failures = 0;
     }
     return result;
   } catch (error) {
     this.failures++;
     this.lastFailTime = Date.now();

     if (this.failures >= 5) {
       this.state = 'open';
     }

     throw error;
   }
 }
}

// Retry mechanism with exponential backoff
async function retryWithBackoff<T>(
 operation: () => Promise<T>,
 maxRetries = 3
): Promise<T> {
 let lastError;
  for (let i = 0; i < maxRetries; i++) {
   try {
     return await operation();
   } catch (error) {
     lastError = error;
     const delay = Math.min(1000 * Math.pow(2, i), 10000);
```

```
      await new Promise(resolve => setTimeout(resolve, delay));
    }
  }
  throw lastError;
}
```

4. Testing Patterns
```typescript
// Integration test example
describe('MCP Integration Tests', () => {
  beforeAll(async () => {
    // Initialize MCP connections
    await initializeMCPServers();
  });
  afterAll(async () => {
    // Cleanup
    await cleanupMCPConnections();
  });
  test('Redis caching works correctly', async () => {
    const key = 'test:key';
    const value = { data: 'test' };

    await mcp.redis.set(key, JSON.stringify(value));
    const retrieved = await mcp.redis.get(key);

    expect(JSON.parse(retrieved)).toEqual(value);
  });
  test('Supabase persistence handles concurrent writes', async () => {
    const promises = Array(10).fill(null).map((_, i) =>
      mcp.supabase
        .from('test_table')
        .insert({ id: i, data: `test-${i}` })
    );

    const results = await Promise.all(promises);
    expect(results.every(r => r.error === null)).toBe(true);
  });
});
```

🚀 Deployment Considerations

````
### Environment Configuration
```typescript
// config/mcp.config.ts
export const MCPConfig = {
 redis: {
   host: process.env.REDIS_HOST || 'localhost',
   port: parseInt(process.env.REDIS_PORT || '6379'),
   password: process.env.REDIS_PASSWORD,
   tls: process.env.NODE_ENV === 'production'
 },
 supabase: {
   url: process.env.SUPABASE_URL!,
   anonKey: process.env.SUPABASE_ANON_KEY!,
   serviceKey: process.env.SUPABASE_SERVICE_KEY!
 },
 github: {
   token: process.env.GITHUB_TOKEN!,
   owner: 'clduab11',
   repo: 'gemini-flow'
 },
 // ... other configurations
};

// Validate configuration on startup
function validateMCPConfig() {
 const required = [
   'SUPABASE_URL',
   'SUPABASE_SERVICE_KEY',
   'GITHUB_TOKEN'
 ];
  const missing = required.filter(key => !process.env[key]);
  if (missing.length > 0) {
   throw new Error(`Missing required environment variables: ${missing.join(', ')}`);
 }
}
```

### Performance Monitoring
```typescript
// Monitor MCP operations
class MCPMonitor {
````
````

```
 private metrics: Map<string, any[]> = new Map();
  async trackOperation<T>(
   server: string,
   operation: string,
   fn: () => Promise<T>
 ): Promise<T> {
   const start = performance.now();
   let success = true;

   try {
     const result = await fn();
     return result;
   } catch (error) {
     success = false;
     throw error;
   } finally {
     const duration = performance.now() - start;

     if (!this.metrics.has(server)) {
       this.metrics.set(server, []);
     }

     this.metrics.get(server)!.push({
       operation,
       duration,
       success,
       timestamp: Date.now()
     });

     // Alert if operation is slow
     if (duration > 1000) {
       console.warn(`Slow MCP operation: ${server}.${operation} took ${duration}ms`);
     }
   }
 }
  getMetrics(server?: string) {
   if (server) {
     return this.metrics.get(server) || [];
   }
   return Object.fromEntries(this.metrics);
 }
}
```

```

## 📚 Additional Resources

- [MCP Protocol Specification](https://modelcontextprotocol.org)
- [Gemini-Flow Architecture](./docs/architecture/ARCHITECTURE.md)
- [Quantum Computing Integration Guide](./docs/quantum/QUANTUM-INTEGRATION.md)
- [Swarm Orchestration Patterns](./docs/swarm/ORCHESTRATION-PATTERNS.md)

## 🤝 Contributing Guidelines

When contributing to Gemini-Flow using GitHub Copilot Pro:

1. **Always use MCP servers** for external integrations
2. **Follow TypeScript best practices** with strict typing
3. **Implement comprehensive error handling** for all MCP operations
4. **Write tests** for new functionality
5. **Document** your MCP integration patterns
6. **Optimize for performance** - batch operations where possible
7. **Ensure security** - never expose credentials in code

Remember: The goal is to create a robust, scalable, and maintainable system that leverages the full power of MCP servers while maintaining the high-performance standards of Gemini-Flow.

---

*This document is optimized for GitHub Copilot Pro to understand and contribute effectively to the Gemini-Flow project using MCP server integrations.*
```