Purple Squirrel Playbook

Strategic Frameworks for Modern Development

Part I: AI/LLM Development

Chapter 1: The Claude Advantage

The landscape of software development has fundamentally shifted. AI assistants are no longer experimental toys—they're force multipliers that can 10x your output when wielded correctly.

Why Claude?

Claude represents a new class of AI assistant: one that can reason about complex problems, maintain context across long conversations, and generate production-quality code. Unlike simple autocomplete tools, Claude thinks.

Key Differentiators:

Capability	Impact
Extended Context	Handle entire codebases, not just snippets
Agentic Behavior	Execute multi-step tasks autonomously
Tool Use	Interact with file systems, APIs, and databases
Reasoning	Explain decisions, catch edge cases

The Mental Model

Think of Claude as a brilliant junior developer with perfect recall but no institutional knowledge. Your job is to:

1. **Provide Context** - Claude doesn't know your codebase, conventions, or constraints
2. **Set Constraints** - Be explicit about what you want and don't want
3. **Verify Output** - Trust but verify; AI makes confident mistakes
4. **Iterate Rapidly** - The cost of asking is near-zero

When to Use Claude

High-Value Tasks:

- Boilerplate generation
- Test writing
- Documentation
- Code refactoring
- Debugging complex issues
- Learning new frameworks

Low-Value Tasks (Do Yourself):

- Critical security code (review AI output carefully)
- Highly domain-specific logic
- Performance-critical hot paths

Chapter 2: Prompt Engineering Fundamentals

The quality of your output is directly proportional to the quality of your input. Prompt engineering isn't magic—it's clear communication.

The Anatomy of a Great Prompt

```
[CONTEXT] What Claude needs to know
[TASK] What you want done
[FORMAT] How you want it delivered
[CONSTRAINTS] What to avoid or include
```

Example: Bad vs Good

Bad Prompt:

> *Write a function to validate emails*

Good Prompt:

> *Write a TypeScript function called `validateEmail` that:*
>
> - *Takes a string input*
> - *Returns `{ valid: boolean, reason?: string }`*
> - *Checks for @ symbol, valid domain, no spaces*
> - *Include JSDoc comments*
> - *Use no external dependencies*

The Golden Rules

1. **Be Specific** - Vague inputs produce vague outputs
2. **Show Examples** - One example is worth 100 words of explanation
3. **State the Obvious** - What's obvious to you isn't to Claude
4. **Iterate in Conversation** - Refine through dialogue

Advanced Techniques

Chain of Thought:

> *Before writing the code, explain your approach step by step, then implement.*

Role Assignment:

> *You are a senior security engineer reviewing this code for vulnerabilities...*

Structured Output:

> *Return your response as JSON with keys: summary, issues, recommendations*

Chapter 3: Building Agentic Systems

Agents are AI systems that can take actions, not just generate text. They represent the next evolution of AI-assisted development.

The Agent Loop

```
OBSERVE → THINK → ACT → OBSERVE → ...
```

1. **Observe** - Read files, check status, gather information
2. **Think** - Reason about what to do next
3. **Act** - Execute commands, write files, make API calls
4. **Repeat** - Continue until goal is achieved

Agent Architecture Patterns

Single Agent:

```
User Request → Agent → Tools → Result
```

Multi-Agent Orchestration:

```
User Request → Coordinator Agent
                        ↓
      ┌─────────────────┼─────────────────┐
      ↓                 ↓                 ↓
Research Agent  Code Agent    Review Agent
      ↓                 ↓                 ↓
      └─────────────────┼─────────────────┘
                        ↓
               Final Result
```

Tool Design Principles

1. **Single Responsibility** - One tool, one job
2. **Clear Interfaces** - Obvious inputs and outputs
3. **Error Handling** - Return meaningful errors, not crashes
4. **Idempotency** - Safe to retry

Example: File Editing Tool

```python
def edit_file(path: str, old_text: str, new_text: str) -> dict:
    """
    Replace old_text with new_text in the specified file.

    Returns:
        {"success": bool, "message": str}
    """
    content = read_file(path)
    if old_text not in content:
        return {"success": False, "message": "Text not found"}

    new_content = content.replace(old_text, new_text, 1)
    write_file(path, new_content)
    return {"success": True, "message": "Edit applied"}
```

Chapter 4: Claude Code Workflows

Claude Code is a CLI tool that brings Claude's capabilities directly into your terminal. Master these workflows to maximize productivity.

Essential Commands

```
# Start interactive session
claude

# Ask a question
claude "How do I implement auth in this project?"

# Resume previous conversation
claude --resume

# Run in non-interactive mode
claude -p "Write tests for user.py"
```

Project Integration

Create a `CLAUDE.md` file in your project root:

```
# Project Context

## Tech Stack
- Python 3.11 with FastAPI
- PostgreSQL with SQLAlchemy
- React 18 with TypeScript

## Conventions
- Use snake_case for Python, camelCase for TypeScript
- All API endpoints return JSON with `data` and `error` keys
- Tests use pytest with fixtures in conftest.py

## Important Files
- `src/core/auth.py` - Authentication logic
- `src/api/routes/` - API endpoints
- `tests/` - Test files mirror src structure
```

Workflow: Feature Development

1. **Plan** - Ask Claude to outline the approach
2. **Scaffold** - Generate file structure and boilerplate
3. **Implement** - Build feature incrementally
4. **Test** - Generate and run tests
5. **Review** - Ask Claude to review for issues

Workflow: Debugging

```
claude "I'm getting this error: [paste error]
```

```
The relevant code is in src/api/users.py.
What's causing this and how do I fix it?"
```

Part II: Startup & Product Strategy

Chapter 5: From Idea to MVP in 48 Hours

Speed is a feature. The faster you can validate an idea, the more ideas you can test, the more likely you are to find one that works.

The 48-Hour Framework

Hour 0-4: Define

- One sentence: What problem does this solve?
- Who has this problem? (Be specific)
- How are they solving it today?

Hour 4-12: Design

- Sketch the core flow (3-5 screens max)
- Define the data model
- Choose the stack (use what you know)

Hour 12-36: Build

- Core feature only—nothing else
- Use templates, boilerplate, AI assistance
- Skip: auth, payments, admin panels (for now)

Hour 36-48: Ship

- Deploy to production
- Create a landing page
- Get it in front of 5 real users

Technology Choices for Speed

Need	Choice	Why
Frontend	Next.js or Remix	Full-stack, fast setup
Backend	FastAPI or Express	Minimal boilerplate
Database	Supabase or PlanetScale	Managed, instant setup
Hosting	Vercel or Railway	Deploy in seconds
Auth	Clerk or Auth.js	Don't build auth

The MVP Checklist

- [ ] Solves one problem well
- [ ] Can be explained in one sentence
- [ ] Works on mobile

- [ ] Has a way to collect feedback
- [ ] Has basic error handling
- [ ] Loads in under 3 seconds

Chapter 6: Technical Architecture Decisions

Early architecture decisions compound. Choose wisely, but don't over-engineer.

The Decision Framework

For each decision, ask:

1. **What's the cost of being wrong?** (Reversibility)
2. **What's the cost of deciding later?** (Delay impact)
3. **What do we know now vs. later?** (Information asymmetry)

High-cost, low-information = Delay if possible **Low-cost, high-information** = Decide now

Monolith vs. Microservices

Start with a Monolith. Always.

Microservices solve organizational problems (multiple teams, different deployment cadences), not technical problems. If you're a small team, microservices are overhead with no benefit.

The path:

```
Monolith → Modular Monolith → Extract Services (when needed)
```

Database Selection

Type	Use When	Examples
PostgreSQL	Default choice, relational data	Users, orders, products
MongoDB	Flexible schemas, documents	CMS content, logs
Redis	Caching, sessions, queues	Rate limiting, real-time
SQLite	Embedded, edge, simple apps	Mobile apps, prototypes

Rule: PostgreSQL until you have a specific reason not to.

The Scaling Playbook

1. **Measure First** - You can't optimize what you don't measure
2. **Cache Aggressively** - Fastest code is code that doesn't run
3. **Database Indexes** - 90% of performance issues are here
4. **CDN Everything Static** - Images, CSS, JS
5. **Queue Background Work** - Don't make users wait

Chapter 7: Scaling Without Breaking

Growth is the goal, but uncontrolled growth breaks systems. Scale deliberately.

The Three Bottlenecks

1. **Database** - Queries get slow, connections max out
2. **Compute** - CPU/memory limits hit
3. **Network** - Bandwidth, latency, timeouts

Database Scaling Ladder

```
Single DB → Read Replicas → Sharding → Distributed DB
    ↑           ↑              ↑            ↑
  1K users   100K users     1M users    10M+ users
```

Caching Strategy

```
Request → Cache Hit? → Yes → Return Cached
              ↓
             No
              ↓
         Database Query
              ↓
         Cache Result
              ↓
         Return Fresh
```

Cache Invalidation Patterns:

- **TTL (Time-to-Live)** - Expire after X seconds
- **Write-Through** - Update cache on every write
- **Event-Driven** - Invalidate on specific events

Load Balancing

```
                    Load Balancer
                         ↓
            ┌────────────┼────────────┐
            ↓            ↓            ↓
        Server 1     Server 2     Server 3
```

Strategies:

- **Round Robin** - Simple, even distribution
- **Least Connections** - Route to least busy server
- **IP Hash** - Same user hits same server (session affinity)

Chapter 8: The Solo Developer's Toolkit

You don't need a team to build great products. You need leverage.

Force Multipliers

1. **AI Coding Assistants** - 2-5x code output
2. **Managed Services** - Don't run your own Postgres
3. **No-Code/Low-Code Tools** - For non-core features

4. **Templates & Boilerplate** - Don't start from scratch
5. **Open Source** - Stand on the shoulders of giants

The Solo Stack

```
Frontend:    Next.js (React + API routes)
Database:    Supabase (Postgres + Auth + Storage)
Payments:    Stripe
Email:       Resend or Postmark
Hosting:     Vercel
Analytics:   Plausible or PostHog
Monitoring:  Sentry
```

Total Monthly Cost: ~$50-100 until significant scale

Time Management

Task Type	Automate?	Outsource?	Do Yourself?
Core Product	No	No	Yes
DevOps	Yes	Maybe	Minimize
Customer Support	Yes (FAQs)	Maybe	Initially Yes
Marketing	Some	Maybe	Learn basics
Legal/Accounting	No	Yes	No

The 80/20 of Solo Development

Spend 80% of time on:

- Features users request
- Performance users notice
- Bugs users report

Spend 20% of time on:

- Technical debt
- Tooling improvements
- Learning new things

Part III: Crypto/Web3 Operations

Chapter 9: Blockchain Security Fundamentals

In crypto, security isn't a feature—it's the product. One mistake can be catastrophic and irreversible.

The Threat Landscape

Threat	Description	Mitigation
Private Key Theft	Attacker obtains signing keys	Hardware wallets, ephemeral keys

Smart Contract Exploits	Bugs in on-chain code	Audits, formal verification
Social Engineering	Phishing, impersonation	Security training, verification
Supply Chain	Compromised dependencies	Dependency auditing, minimal deps

Key Security Principles

1. **Minimize Key Exposure** - Keys should exist for the shortest time possible
2. **Defense in Depth** - Multiple layers of security
3. **Assume Breach** - Design for when (not if) compromise occurs
4. **Verify Everything** - Never trust, always verify

The ColdStar Philosophy

Traditional hardware wallets create a permanent target. ColdStar inverts this:

```
Traditional:  Permanent Key → Permanent Risk
ColdStar:     Ephemeral Key → Ephemeral Risk (~100µs)
```

Key only exists during signing:

1. Decrypt from USB into locked RAM
2. Sign transaction
3. Immediately zeroize memory

Operational Security Checklist

- [ ] Keys generated on air-gapped machine
- [ ] Backup seed phrase stored securely (not digitally)
- [ ] Hardware wallet for hot operations
- [ ] Multi-sig for high-value treasuries
- [ ] Transaction simulation before signing
- [ ] Allowlist for contract interactions

Chapter 10: Wallet Architecture

The architecture of your wallet system determines your security ceiling. Design it right from the start.

Wallet Types

Type	Use Case	Security	Convenience
Hot Wallet	Daily operations	Lower	High
Cold Wallet	Long-term storage	High	Lower
Hardware Wallet	Interactive cold storage	High	Medium
Multi-sig	Treasuries, DAOs	Highest	Lowest

The Tiered Architecture

```
┌─────────────────────────────────────┐
│           Hot Wallet (5%)           │  Daily ops, automated
│          Funded as needed           │
└─────────────────────────────────────┘
                 ↑
┌─────────────────────────────────────┐
│          Warm Wallet (15%)          │  Weekly operations
│        Hardware wallet based        │
└─────────────────────────────────────┘
                 ↑
┌─────────────────────────────────────┐
│          Cold Storage (80%)         │  Long-term, multi-sig
│    Air-gapped, geographic split     │
└─────────────────────────────────────┘
```

Key Generation Best Practices

1. **Use Hardware Entropy** - Never use Math.random() or similar
2. **Air-Gapped Generation** - No network connection during generation
3. **Verify Randomness** - Multiple entropy sources when possible
4. **Secure Backup** - Physical backup, multiple locations

Memory Security

```rust
// Rust example: Secure key handling
use zeroize::Zeroize;

struct SecretKey {
    bytes: [u8; 32],
}

impl Drop for SecretKey {
    fn drop(&mut self) {
        self.bytes.zeroize(); // Overwrite with zeros
    }
}
```

Chapter 11: DeFi Integration Patterns

DeFi protocols are composable, but that composability creates complexity. Navigate it safely.

The DeFi Stack

```
┌─────────────────────────────────────┐
│          Your Application           │
└─────────────────────────────────────┘
                 ↓
┌─────────────────────────────────────┐
│     Aggregators (1inch, Jupiter)    │
└─────────────────────────────────────┘
                 ↓
```

```
┌─────────────────────────────────────┐
│         DEXs (Uniswap, Raydium)     │
└─────────────────────────────────────┘
                  ↓
┌─────────────────────────────────────┐
│        Tokens (ERC-20, SPL)         │
└─────────────────────────────────────┘
                  ↓
┌─────────────────────────────────────┐
│          Blockchain (ETH, SOL)      │
└─────────────────────────────────────┘
```

Common Integration Patterns

Swapping Tokens:

```
# Using Jupiter API (Solana)
quote = get_quote(
    input_mint="SOL",
    output_mint="USDC",
    amount=1_000_000_000,  # 1 SOL in lamports
    slippage_bps=50        # 0.5% slippage
)

transaction = get_swap_transaction(quote)
signed = sign_transaction(transaction, keypair)
send_transaction(signed)
```

Staking:

```
# Liquid staking pattern
stake_instruction = create_stake_instruction(
    amount=amount,
    stake_pool=JITO_POOL,
    user=user_pubkey
)
# User receives jSOL in return
```

Risk Management

Risk	Mitigation
Slippage	Set max slippage, use limit orders
Impermanent Loss	Understand LP math, monitor positions
Smart Contract Risk	Stick to audited protocols
Oracle Manipulation	Use multiple price sources
Rug Pulls	Verify contract ownership, check liquidity locks

Chapter 12: Smart Contract Best Practices

Smart contracts are immutable. Bugs are permanent. Test relentlessly.

Development Workflow

```
Write → Test → Audit → Deploy to Testnet → Audit → Deploy to Mainnet
          ↑                    ↑
          └────────────────────┘
               Iterate
```

Testing Hierarchy

1. **Unit Tests** - Individual function behavior
2. **Integration Tests** - Multi-contract interactions
3. **Fuzz Testing** - Random inputs to find edge cases
4. **Formal Verification** - Mathematical proof of correctness

Common Vulnerabilities

Reentrancy:

```
// VULNERABLE
function withdraw() external {
    uint amount = balances[msg.sender];
    (bool success,) = msg.sender.call{value: amount}("");
    balances[msg.sender] = 0; // Too late!
}

// SAFE
function withdraw() external {
    uint amount = balances[msg.sender];
    balances[msg.sender] = 0; // Update first
    (bool success,) = msg.sender.call{value: amount}("");
}
```

Integer Overflow (pre-0.8.0):

```
// Use SafeMath or Solidity 0.8+
uint256 result = a + b; // Auto-reverts on overflow in 0.8+
```

Upgrade Patterns

Pattern	Pros	Cons
Immutable	Simple, trustless	Can't fix bugs
Proxy	Upgradeable	Trust admin, complexity
Diamond	Modular upgrades	Complex, more surface area

Recommendation: Start immutable. Use proxy only if you have a clear upgrade governance model.

Afterword

The playbook is just the beginning. True mastery comes from application—building, shipping, failing, and iterating.

The tools change. The frameworks evolve. The chains fork. But the fundamentals remain:

- **Solve real problems** for real people
- **Ship early** and often
- **Security first** in everything you build
- **Leverage AI** to multiply your output
- **Stay curious** and keep learning

Now close this book and go build something.

Purple Squirrel Media Engineering Excellence