Purple Squirrel Playbook

Strategic Frameworks for Modern Development

By Purple Squirrel Media

Preface

This book is for developers who want to build better software faster. Whether you're a solo founder shipping an MVP, a senior engineer scaling a platform, or an operator managing blockchain infrastructure, you'll find actionable frameworks here.

We cover three domains: AI-assisted development, startup engineering, and crypto operations. Each chapter gives you mental models and practical patterns you can apply immediately.

Let's get started.

Part I. AI/LLM Development

The landscape of software development has fundamentally shifted. AI assistants are no longer experimental—they're force multipliers that can dramatically increase your output when wielded correctly. This section teaches you how.

Chapter 1. The Claude Advantage

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.

This matters because the way you work with Claude is different from how you might use simpler tools. You're not just getting suggestions—you're collaborating with a capable assistant that can understand your entire codebase, debug complex issues, and even execute multi-step tasks autonomously.

Key Capabilities

What makes Claude different from other AI tools?

Extended Context. Claude can handle entire codebases, not just snippets. You can paste thousands of lines of code and ask questions about the whole thing.

Agentic Behavior. Claude can execute multi-step tasks autonomously. Give it a goal, and it figures out the steps.

Tool Use. Claude can interact with file systems, APIs, and databases. It's not just generating text—it's taking actions.

Reasoning. Claude explains its decisions and catches edge cases. When something doesn't look right, it tells you.

The Mental Model

Think of Claude as a brilliant junior developer with perfect recall but no institutional knowledge. This mental model helps you understand how to work with it effectively.

Your job is to provide context. Claude doesn't know your codebase, your team's conventions, or your business constraints. The more context you give, the better the output.

You also need to set constraints. Be explicit about what you want and what you don't want. "Write a function" is vague. "Write a TypeScript function that validates email addresses and returns a result object with a boolean and optional error message" is specific.

Always verify output. Trust but verify—AI makes confident mistakes. Just because Claude sounds certain doesn't mean it's right. Review the code, test it, and understand what it does before shipping.

Finally, iterate rapidly. The cost of asking is near-zero. If the first response isn't quite right, refine your request. Conversation is cheap.

When to Use Claude

Claude excels at certain tasks while others are better done yourself.

High-value tasks where Claude shines include boilerplate generation, test writing, documentation, code refactoring, debugging complex issues, and learning new frameworks. These are time-consuming but relatively mechanical—perfect for AI assistance.

Tasks to do yourself include critical security code (always review AI output carefully for security implications), highly domain-specific logic that requires deep business knowledge, and performance-critical hot paths where every optimization matters.

> ***Note:*** *The goal isn't to replace your judgment—it's to augment your capabilities. Use Claude for the mechanical work so you can focus on the creative and critical decisions.*

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. This chapter teaches you how to communicate effectively with AI.

The Anatomy of a Great Prompt

Every great prompt has four components:

Context tells Claude what it needs to know. What's the project? What language? What constraints exist?

Task tells Claude what you want done. Be specific about the deliverable.

Format tells Claude how you want it delivered. JSON? Markdown? Code with comments?

Constraints tell Claude what to avoid or include. No external dependencies? Must handle edge cases?

Here's the difference between a bad prompt and a good one:

Bad: "Write a function to validate emails"

Good: "Write a TypeScript function called `validateEmail` that takes a string input and returns `{ valid: boolean, reason?: string }`. Check for @ symbol, valid domain format, and no spaces. Include JSDoc comments. Use no external dependencies."

The good prompt is specific about the language, function name, return type, validation rules, documentation requirements, and constraints. Claude can execute on this immediately without asking clarifying questions.

The Golden Rules

Four rules will improve every prompt you write:

Be specific. Vague inputs produce vague outputs. If you want TypeScript, say TypeScript. If you want error handling, describe what kind.

Show examples. One example is worth a hundred words of explanation. If you want output in a certain format, show that format.

State the obvious. What's obvious to you isn't obvious to Claude. It doesn't know your conventions, your preferences, or your constraints unless you tell it.

Iterate in conversation. Refine through dialogue. The first response rarely needs to be the last. Build on it.

Advanced Techniques

Once you've mastered the basics, these techniques will take your prompts further:

Chain of Thought asks Claude to explain its reasoning before giving an answer. "Before writing the code, explain your approach step by step, then implement." This catches errors in logic before they become errors in code.

Role Assignment puts Claude in a specific mindset. "You are a senior security engineer reviewing this code for vulnerabilities" will produce different (and often better) output than a generic request.

Structured Output specifies the exact format you need. "Return your response as JSON with keys: summary, issues, recommendations" ensures you get data you can parse programmatically.

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. This chapter explains how they work and how to build them.

The Agent Loop

Every agent follows the same fundamental loop:

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

Observe means gathering information—reading files, checking status, making API calls to understand the current state.

Think means reasoning about what to do next. Given what I know, what's the best action?

Act means executing—writing files, running commands, making changes.

Repeat means continuing the loop until the goal is achieved.

This loop is powerful because it allows the agent to adapt. If an action fails, the agent observes the failure and adjusts its approach.

Architecture Patterns

There are two main patterns for building agents:

Single Agent is the simplest pattern. One agent receives a request, uses tools to accomplish the task, and returns a result:

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

This works well for focused tasks like "refactor this function" or "write tests for this module."

Multi-Agent Orchestration uses multiple specialized agents coordinated by a central agent:

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

This works well for complex tasks like "implement this feature end-to-end" where different aspects benefit from different approaches.

Tool Design Principles

Good tools make good agents. Follow these principles:

Single Responsibility. Each tool does one thing well. A file editing tool edits files. A search tool searches. Don't combine unrelated functionality.

Clear Interfaces. Inputs and outputs should be obvious. Use descriptive parameter names and return structured data.

Error Handling. Return meaningful errors, not crashes. If a file doesn't exist, return an error message explaining that—don't throw an exception that crashes the agent.

Idempotency. Tools should be safe to retry. If an operation fails partway through, running it again should work correctly.

Here's an example of a well-designed 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 in file"}

    new_content = content.replace(old_text, new_text, 1)
```

```
    write_file(path, new_content)
    return {"success": True, "message": "Edit applied successfully"}
```

This tool has a single responsibility, clear inputs and outputs, proper error handling, and is safe to retry.

Chapter 4. Claude Code Workflows

Claude Code is a CLI tool that brings Claude's capabilities directly into your terminal. This chapter shows you practical workflows for using it effectively.

Essential Commands

Start an interactive session:

```
claude
```

Ask a question directly:

```
claude "How do I implement auth in this project?"
```

Resume a previous conversation:

```
claude --resume
```

Run in non-interactive mode for scripts:

```
claude -p "Write tests for user.py"
```

Project Integration

Create a `CLAUDE.md` file in your project root to give Claude permanent context about your project:

```
# 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
```

Claude reads this file automatically, so you don't have to repeat project context in every conversation.

Feature Development Workflow

When building a new feature, follow this workflow:

Plan. Ask Claude to outline the approach before writing code. "I need to add user authentication. What's the best approach given our current stack?"

Scaffold. Generate the file structure and boilerplate. "Create the files I'll need for JWT authentication with refresh tokens."

Implement. Build the feature incrementally. Work through one component at a time, testing as you go.

Test. Generate tests for what you've built. "Write tests for the authentication flow we just implemented."

Review. Ask Claude to review the complete implementation. "Review this authentication code for security issues and edge cases."

Debugging Workflow

When debugging, give Claude the error and relevant context:

```
I'm getting this error: [paste the full error]

The relevant code is in src/api/users.py.

What's causing this and how do I fix it?
```

The more context you provide—error messages, stack traces, relevant code—the faster Claude can identify the problem.

Part II. Startup & Product Strategy

Building a startup is different from building software at an established company. Speed matters more. Resources are limited. This section covers engineering strategy for the startup context.

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. This chapter shows you how to ship an MVP in 48 hours.

The 48-Hour Framework

Hours 0-4: Define. Answer three questions: What problem does this solve? (One sentence.) Who has this problem? (Be specific—not "everyone.") How are they solving it today?

Hours 4-12: Design. Sketch the core flow—three to five screens maximum. Define the data model. Choose your stack (use what you know, not what's trendy).

Hours 12-36: Build. Build the core feature only. Nothing else. Use templates, boilerplate, and AI assistance liberally. Skip authentication, payments, and admin panels for now.

Hours 36-48: Ship. Deploy to production. Create a landing page. Get it in front of five real users.

Technology Choices for Speed

When speed matters, choose technologies that minimize setup time:

For **frontend**, use Next.js or Remix. They're full-stack frameworks with fast setup.

For **backend**, use FastAPI (Python) or Express (Node). Minimal boilerplate, fast iteration.

For **database**, use Supabase or PlanetScale. Managed services with instant setup.

For **hosting**, use Vercel or Railway. Deploy in seconds, not hours.

For **auth**, use Clerk or Auth.js. Don't build authentication yourself for an MVP.

The MVP Checklist

Before you ship, verify:

- It solves one problem well
- It can be explained in one sentence
- It works on mobile
- It has a way to collect feedback
- It has basic error handling
- It loads in under 3 seconds

If you can check all six, ship it.

Chapter 6. Technical Architecture Decisions

Early architecture decisions compound. A good decision saves you time for years. A bad one creates drag on every feature you build. This chapter helps you decide wisely.

The Decision Framework

For each technical decision, ask three questions:

What's the cost of being wrong? Some decisions are easily reversible (which logging library to use). Others are expensive to change (which database to use). Spend more time on expensive decisions.

What's the cost of deciding later? Sometimes waiting gives you more information. Sometimes waiting creates technical debt. Know which situation you're in.

What do you know now vs. what you'll know later? If you'll have much better information in a month, consider waiting. If not, decide now.

Monolith vs. Microservices

Start with a monolith. Always.

Microservices solve organizational problems—multiple teams, different deployment cadences, different scaling requirements. They don't solve technical problems better than a well-structured monolith.

If you're a small team, microservices are overhead with no benefit. You're adding network calls, deployment complexity, and operational burden without any upside.

The path forward: Monolith → Modular Monolith → Extract Services (when you actually need to).

Database Selection

PostgreSQL should be your default choice. It handles relational data well, scales further than most people realize, and has excellent tooling.

Use MongoDB when you need flexible schemas and your data is genuinely document-shaped. CMS content and logs are good examples.

Use Redis for caching, sessions, and queues. It's fast and simple.

Use SQLite for embedded databases, edge computing, or simple applications where you don't need a server.

The rule: PostgreSQL until you have a specific reason not to.

Chapter 7. Scaling Without Breaking

Growth is the goal, but uncontrolled growth breaks systems. This chapter covers how to scale deliberately without creating chaos.

The Three Bottlenecks

Every scaling problem comes down to one of three bottlenecks:

Database. Queries get slow, connections max out. This is the most common bottleneck.

Compute. CPU or memory limits get hit. Usually happens with compute-intensive operations.

Network. Bandwidth limits or latency issues. More common with distributed systems.

Identify which bottleneck you're hitting before trying to fix it. The solutions are different.

Database Scaling Ladder

Scale your database in stages:

Single database handles most applications up to a few thousand users. Don't optimize prematurely.

Read replicas help when read traffic vastly exceeds write traffic. Direct reads to replicas, writes to primary.

Sharding splits data across multiple databases. Complex to implement, necessary at scale.

Distributed databases like CockroachDB or Spanner handle massive scale but add operational complexity.

Move up the ladder only when you need to. Each step adds complexity.

Caching Strategy

The fastest code is code that doesn't run. Caching lets you skip expensive operations.

```
Request → Cache Hit? → Yes → Return Cached
              ↓
             No
              ↓
         Compute/Query
              ↓
         Cache Result
```

```
              ↓
         Return Fresh
```

For cache invalidation, you have three main patterns:

TTL (Time-to-Live) expires cached data after a set time. Simple but can serve stale data.

Write-Through updates the cache whenever you write to the database. Always fresh, but more complex.

Event-Driven invalidates cache based on specific events. Most flexible, most complex.

Chapter 8. The Solo Developer's Toolkit

You don't need a team to build great products. You need leverage. This chapter covers how to maximize your output as a solo developer.

Force Multipliers

Five things multiply your effectiveness:

AI coding assistants increase code output 2-5x for many tasks. Use them heavily.

Managed services mean you don't run your own database, don't manage your own servers, don't handle your own email delivery. Pay for operations you'd otherwise do yourself.

No-code/low-code tools handle non-core features. Use Zapier for integrations, Retool for admin panels, Typeform for surveys.

Templates and boilerplate mean you don't start from scratch. Find good starting points and customize.

Open source means you stand on the shoulders of giants. Don't build what others have built well.

The Solo Stack

Here's a complete stack that one person can operate:

- **Frontend:** Next.js (React with API routes built in)
- **Database:** Supabase (Postgres with auth and storage included)
- **Payments:** Stripe
- **Email:** Resend or Postmark
- **Hosting:** Vercel
- **Analytics:** Plausible or PostHog
- **Monitoring:** Sentry

Total monthly cost: $50-100 until significant scale. All managed, all reliable.

Time Management

As a solo developer, your time is your only resource. Spend it wisely.

Do yourself: Core product work. This is where your unique value lies.

Automate: DevOps, testing, deployment. Set it up once, let it run.

Outsource: Legal, accounting, and tasks where experts are much better than you.

Skip: Features nobody asked for. Premature optimization. Perfect code for throwaway prototypes.

The 80/20 rule: Spend 80% of your time on features users request, bugs users report, and performance users notice. Spend 20% on technical debt, tooling, and learning.

Part III. Crypto/Web3 Operations

Blockchain systems have unique security requirements. A bug in a web app loses data; a bug in a smart contract loses money. This section covers how to operate safely in crypto.

Chapter 9. Blockchain Security Fundamentals

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

The Threat Landscape

Four categories of threats matter most:

Private key theft is the most direct attack. If someone gets your signing keys, they control your funds. Mitigation: hardware wallets, ephemeral keys, multi-signature schemes.

Smart contract exploits target bugs in on-chain code. Reentrancy, integer overflow, access control failures. Mitigation: audits, formal verification, battle-tested patterns.

Social engineering targets humans, not systems. Phishing, impersonation, fake websites. Mitigation: security training, verification procedures, healthy paranoia.

Supply chain attacks compromise your dependencies. A malicious package update, a compromised build system. Mitigation: dependency auditing, minimal dependencies, reproducible builds.

Security Principles

Four principles guide secure blockchain operations:

Minimize key exposure. Keys should exist for the shortest time possible. The ColdStar approach: keys exist in RAM for microseconds during signing, then are immediately destroyed.

Defense in depth. Multiple layers of security. If one fails, others protect you. Don't rely on any single control.

Assume breach. Design for when—not if—compromise occurs. Limit blast radius. Have recovery procedures.

Verify everything. Never trust, always verify. Check signatures, validate addresses, confirm transactions before signing.

Operational Security Checklist

Before going live with any blockchain system:

- 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. Get it right from the start.

Wallet Types

Hot wallets are connected to the internet. Fast and convenient, but higher risk. Use for daily operations.

Cold wallets are offline. Higher security, less convenient. Use for long-term storage.

Hardware wallets are dedicated devices for key storage. Good balance of security and usability.

Multi-sig wallets require multiple signatures to authorize transactions. Highest security for treasuries and DAOs.

The Tiered Architecture

For significant funds, use a tiered architecture:

Hot wallet (5% of funds): Daily operations, automated transactions. Funded as needed from warm wallet.

Warm wallet (15% of funds): Weekly operations, manual transactions. Hardware wallet based.

Cold storage (80% of funds): Long-term holdings. Air-gapped, multi-signature, geographically distributed.

This limits your exposure. If the hot wallet is compromised, you lose 5%, not 100%.

Key Generation

Generate keys correctly:

Use hardware entropy. Never use `Math.random()` or similar weak sources. Use `/dev/urandom` or a hardware random number generator.

Air-gapped generation. No network connection during key generation. Ever.

Verify randomness. Use multiple entropy sources when possible. Don't trust any single source completely.

Secure backup. Physical backup in multiple locations. Never store seed phrases digitally.

Chapter 11. DeFi Integration Patterns

DeFi protocols are composable—you can combine them like building blocks. This composability creates power and complexity. This chapter shows you how to navigate it.

The DeFi Stack

From bottom to top:

Blockchain layer: Ethereum, Solana, etc. The foundation.

Token layer: ERC-20, SPL tokens. The assets you're moving.

Protocol layer: Uniswap, Aave, Compound. The applications.

Aggregator layer: 1inch, Jupiter. Services that find the best routes across protocols.

Application layer: Your code that ties it all together.

Common Patterns

Token swaps exchange one token for another. Use aggregators for best prices:

```
quote = get_quote(
    input_mint="SOL",
    output_mint="USDC",
    amount=1_000_000_000,  # 1 SOL in lamports
    slippage_bps=50        # 0.5% slippage tolerance
)
transaction = get_swap_transaction(quote)
signed = sign_transaction(transaction, keypair)
send_transaction(signed)
```

Staking locks tokens for yield. Understand the lockup period, slashing conditions, and reward mechanism before staking.

Liquidity provision supplies tokens to a pool in exchange for fees. Understand impermanent loss before providing liquidity.

Risk Management

DeFi has specific risks:

Slippage means getting a worse price than expected. Set maximum slippage and use limit orders for large trades.

Impermanent loss affects liquidity providers when token prices change. Understand the math before providing liquidity.

Smart contract risk means the protocol itself could have bugs. Stick to audited, battle-tested protocols.

Oracle manipulation can cause incorrect pricing. Use protocols that rely on multiple price sources.

Chapter 12. Smart Contract Best Practices

Smart contracts are immutable. Once deployed, bugs are permanent. Test relentlessly before deploying.

Development Workflow

Follow this workflow:

Write the contract with security in mind from the start.

Test with unit tests, integration tests, and fuzz testing.

Audit with internal review first, then external auditors for significant contracts.

Deploy to testnet and test again in a realistic environment.

Audit again after any changes.

Deploy to mainnet only when confident.

Common Vulnerabilities

Know these vulnerabilities and how to prevent them:

Reentrancy occurs when a contract calls an external contract that calls back before the first call completes. Prevention: update state before making external calls.

```
// 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 state first
    (bool success,) = msg.sender.call{value: amount}("");
}
```

Integer overflow (in Solidity < 0.8) occurs when arithmetic exceeds type limits. Prevention: use Solidity 0.8+ or SafeMath.

Access control failures occur when functions don't properly restrict who can call them. Prevention: use modifiers consistently, default to restrictive.

Upgrade Patterns

If you need upgradeable contracts:

Immutable is simplest—no upgrades possible. Can't fix bugs, but also can't introduce them post-deployment.

Proxy pattern allows upgrades by delegating to an implementation contract. Adds complexity and trust requirements.

Recommendation: start with immutable contracts. Use proxy pattern only if you have clear upgrade governance and the benefits outweigh the risks.

Afterword

The frameworks in this book are starting points, not destinations. True mastery comes from application—building, shipping, failing, and iterating.

Tools change. Frameworks evolve. Blockchains fork. But fundamentals remain:

Solve real problems for real people. Ship early and often. Put security first in everything you build. Use AI to multiply your output. Stay curious and keep learning.

Now close this book and go build something.

Purple Squirrel Media