Purple Squirrel Cookbook

Solutions and Patterns for Every Problem

By Purple Squirrel Media

Preface

A cookbook is different from other technical books. You don't read it cover to cover. You come to it with a specific problem, find the recipe that addresses it, and adapt the solution to your needs.

Each recipe in this book follows a consistent pattern. We state the problem clearly, present a complete working solution, and explain the key decisions. Code samples are designed to be copied and modified. They prioritize clarity over cleverness and are production-ready rather than simplified for pedagogical purposes.

The recipes are organized into three parts. Part I covers Claude and AI integration patterns, from basic API calls to sophisticated RAG implementations. Part II provides full-stack recipes for authentication, real-time communication, file handling, and payments. Part III addresses DevOps concerns including containerization, CI/CD, monitoring, and database management.

When you find a recipe that solves your problem, don't just copy the code. Read through the explanation to understand why it works. The patterns here are meant to be internalized and adapted, not just applied blindly.

Part I. Claude and AI Recipes

Large language models have become essential tools in modern software development. This section provides practical recipes for integrating Claude into your applications, from simple API calls to complex multi-turn conversations and retrieval-augmented generation.

Chapter 1. Working with the Claude API

The Claude API provides a straightforward interface for building AI-powered applications. These recipes cover the fundamental patterns you'll use in virtually every integration.

Structured Output Extraction

Getting Claude to return data in a format your code can parse requires clear instructions and explicit format specification.

The key insight is that Claude responds well to examples. Rather than describing the format abstractly, show Claude exactly what you want.

```
import anthropic

client = anthropic.Anthropic()

def extract_entities(text: str) -> dict:
    response = client.messages.create(
        model="claude-sonnet-4-20250514",
```

```
        max_tokens=1024,
        messages=[{
            "role": "user",
            "content": f"""Extract entities from this text and return as JSON.

Text: {text}

Return format:
{{
    "people": ["name1", "name2"],
    "companies": ["company1"],
    "dates": ["2024-01-15"],
    "amounts": ["$1,000"]
}}

Return ONLY valid JSON, no explanation."""
        }]
    )

    import json
    return json.loads(response.content[0].text)
```

The format block serves as both instruction and example. By including realistic sample data in the template, you anchor Claude's understanding of what each field should contain.

For complex schemas with nested objects or arrays, provide a complete example response rather than trying to describe the structure in prose. Claude will match the pattern more reliably than it will follow abstract specifications.

Multi-turn Conversation with Memory

A conversational interface requires maintaining context across multiple exchanges. The Claude API is stateless, so your application must manage the conversation history.

```
class Conversation:
    def __init__(self):
        self.client = anthropic.Anthropic()
        self.messages = []
        self.system = "You are a helpful coding assistant."

    def chat(self, user_input: str) -> str:
        self.messages.append({
            "role": "user",
            "content": user_input
        })

        response = self.client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=4096,
            system=self.system,
            messages=self.messages
        )
```

```
        assistant_message = response.content[0].text
        self.messages.append({
            "role": "assistant",
            "content": assistant_message
        })

        return assistant_message

    def clear(self):
        self.messages = []
```

The message list grows with each exchange. When context becomes too large, you have several options. You can summarize earlier messages, remove the oldest exchanges, or start fresh with a summary of the conversation so far.

For production applications, consider persisting the conversation history to a database. This allows users to resume conversations across sessions and provides valuable data for understanding how your application is being used.

Tool Use and Function Calling

Tool use allows Claude to interact with external systems by requesting that your code execute specific functions. This pattern is essential for building agents that can take actions in the world.

```
import anthropic
import json

tools = [
    {
        "name": "get_weather",
        "description": "Get current weather for a location",
        "input_schema": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "City name"
                }
            },
            "required": ["location"]
        }
    }
]

def get_weather(location: str) -> dict:
    # In production, this would call a weather API
    return {"temp": 72, "condition": "sunny"}

def process_with_tools(user_input: str):
    client = anthropic.Anthropic()
```

```
    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=1024,
        tools=tools,
        messages=[{"role": "user", "content": user_input}]
    )

    for block in response.content:
        if block.type == "tool_use":
            if block.name == "get_weather":
                result = get_weather(block.input["location"])
                return continue_with_tool_result(
                    response, block.id, result
                )

    return response.content[0].text
```

The tool schema tells Claude what functions are available and what parameters they accept. When Claude decides to use a tool, your code executes the actual function and sends the result back. This creates a loop where Claude can request multiple tool calls to accomplish complex tasks.

Write clear, specific descriptions for your tools. Claude uses these descriptions to decide when to use each tool, so vague descriptions lead to unreliable behavior.

Chapter 2. Retrieval-Augmented Generation

RAG allows Claude to answer questions using information from your own documents. The pattern involves retrieving relevant content and including it in the prompt context.

Basic RAG Implementation

The core RAG pattern has three steps: embed your documents, find relevant chunks based on the query, and include those chunks in the prompt.

```
from sentence_transformers import SentenceTransformer
import numpy as np

class SimpleRAG:
    def __init__(self):
        self.encoder = SentenceTransformer('all-MiniLM-L6-v2')
        self.documents = []
        self.embeddings = None

    def add_documents(self, docs: list[str]):
        self.documents.extend(docs)
        self.embeddings = self.encoder.encode(self.documents)

    def retrieve(self, query: str, top_k: int = 3) -> list[str]:
        query_embedding = self.encoder.encode([query])[0]
        similarities = np.dot(self.embeddings, query_embedding)
        top_indices = np.argsort(similarities)[-top_k:][::-1]
        return [self.documents[i] for i in top_indices]
```

```
    def answer(self, question: str) -> str:
        relevant_docs = self.retrieve(question)
        context = "\n\n".join(relevant_docs)

        client = anthropic.Anthropic()
        response = client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=1024,
            messages=[{
                "role": "user",
                "content": f"""Answer based on this context:

{context}

Question: {question}

If the answer isn't in the context, say "I don't have that information."""
            }]
        )
        return response.content[0].text
```

This implementation uses cosine similarity to find the most relevant documents. For production use, you would store embeddings in a vector database like Pinecone, Weaviate, or pgvector rather than computing similarities in memory.

The instruction to acknowledge missing information is important. Without it, Claude may hallucinate answers that seem plausible but aren't grounded in your documents.

Automated Code Review

Claude excels at reviewing code changes. By providing the diff and clear review criteria, you can get consistent, thorough feedback on every pull request.

```
def review_code(diff: str, context: str = "") -> dict:
    client = anthropic.Anthropic()

    prompt = f"""Review this code diff and provide feedback.

{f"Context: {context}" if context else ""}

```diff
{diff}
```

Analyze for:

1. Bugs or logic errors
2. Security issues
3. Performance concerns
4. Code style/readability
5. Missing edge cases
```

Return as JSON: {{ "summary": "One sentence overview", "issues": [{{ "severity": "high|medium|low", "line": 42, "issue": "Description", "suggestion": "How to fix" }}], "approved": true/false }}"""

```
response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=2048,
    messages=[{"role": "user", "content": prompt}]
)

import json
return json.loads(response.content[0].text)
```

```
The structured output format makes it easy to integrate code review into your CI
pipeline. You can block merges on high-severity issues or post comments directly on
pull requests.

Providing context about what the code is supposed to do improves review quality
significantly. Claude can identify logic errors much more effectively when it
understands the intent.

# Part II. Full-Stack Recipes

Modern applications require authentication, real-time communication, file handling,
and often payment processing. These recipes provide production-ready patterns for
these common requirements.

## Chapter 3. Authentication

Authentication is a critical security concern. These recipes implement standard
patterns using proven libraries.

### JWT Authentication System

JSON Web Tokens provide stateless authentication. This implementation includes both
access tokens for short-lived authentication and refresh tokens for maintaining
sessions.

```python
from datetime import datetime, timedelta
from typing import Optional
import jwt
from passlib.context import CryptContext

SECRET_KEY = "your-secret-key"  # Use environment variable
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE = timedelta(minutes=15)
REFRESH_TOKEN_EXPIRE = timedelta(days=7)

pwd_context = CryptContext(schemes=["bcrypt"])

def hash_password(password: str) -> str:
```
```

```
    return pwd_context.hash(password)

def verify_password(plain: str, hashed: str) -> bool:
    return pwd_context.verify(plain, hashed)

def create_access_token(user_id: str) -> str:
    expire = datetime.utcnow() + ACCESS_TOKEN_EXPIRE
    payload = {
        "sub": user_id,
        "exp": expire,
        "type": "access"
    }
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def create_refresh_token(user_id: str) -> str:
    expire = datetime.utcnow() + REFRESH_TOKEN_EXPIRE
    payload = {
        "sub": user_id,
        "exp": expire,
        "type": "refresh"
    }
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def verify_token(token: str) -> Optional[dict]:
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except jwt.ExpiredSignatureError:
        return None
    except jwt.InvalidTokenError:
        return None
```

The separation between access and refresh tokens is crucial. Access tokens are short-lived (15 minutes) and used for every authenticated request. Refresh tokens are long-lived (7 days) and used only to obtain new access tokens.

For FastAPI integration, create a dependency that extracts and validates the token:

```
from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer

security = HTTPBearer()

async def get_current_user(credentials = Depends(security)):
    token = credentials.credentials
    payload = verify_token(token)

    if not payload or payload.get("type") != "access":
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid token"
        )
```

```
    return payload["sub"]
```

Store the secret key in environment variables, never in code. In production, consider using asymmetric keys (RS256) so you can verify tokens without access to the signing key.

Chapter 4. Real-Time Communication

Many applications need to push updates to clients as events occur. WebSockets provide bidirectional communication for this use case.

WebSocket Connection Management

Managing multiple WebSocket connections requires careful attention to connection lifecycle and room-based broadcasting.

```
from fastapi import WebSocket
from typing import Dict, Set
import json

class ConnectionManager:
    def __init__(self):
        self.rooms: Dict[str, Set[WebSocket]] = {}

    async def connect(self, websocket: WebSocket, room_id: str):
        await websocket.accept()
        if room_id not in self.rooms:
            self.rooms[room_id] = set()
        self.rooms[room_id].add(websocket)

    def disconnect(self, websocket: WebSocket, room_id: str):
        if room_id in self.rooms:
            self.rooms[room_id].discard(websocket)

    async def broadcast(self, room_id: str, message: dict):
        if room_id not in self.rooms:
            return
        data = json.dumps(message)
        for websocket in self.rooms[room_id]:
            try:
                await websocket.send_text(data)
            except:
                self.rooms[room_id].discard(websocket)

manager = ConnectionManager()
```

The endpoint handles the WebSocket lifecycle and message routing:

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect

app = FastAPI()
```

```python
@app.websocket("/ws/{room_id}")
async def websocket_endpoint(websocket: WebSocket, room_id: str):
    await manager.connect(websocket, room_id)
    try:
        while True:
            data = await websocket.receive_json()
            await manager.broadcast(room_id, {
                "type": "message",
                "data": data
            })
    except WebSocketDisconnect:
        manager.disconnect(websocket, room_id)
```

On the client side, connection is straightforward:

```javascript
const ws = new WebSocket('ws://localhost:8000/ws/room123');

ws.onmessage = (event) => {
  const data = JSON.parse(event.data);
  console.log('Received:', data);
};

ws.send(JSON.stringify({ message: 'Hello!' }));
```

For production, add authentication by verifying a token before accepting the connection. You should also implement heartbeat pings to detect stale connections and automatic reconnection on the client side.

Chapter 5. File Handling

File uploads introduce security and performance concerns. These patterns address validation, storage, and background processing.

Secure File Upload with Processing

Validate files on upload, store them safely, and process asynchronously:

```python
from fastapi import FastAPI, UploadFile, BackgroundTasks
from pathlib import Path
import hashlib
import aiofiles

UPLOAD_DIR = Path("uploads")
MAX_SIZE = 10 * 1024 * 1024  # 10MB
ALLOWED_TYPES = {"image/jpeg", "image/png", "application/pdf"}

async def save_upload(file: UploadFile) -> str:
    if file.content_type not in ALLOWED_TYPES:
        raise ValueError(f"Invalid file type: {file.content_type}")

    content = await file.read()
```

```
    if len(content) > MAX_SIZE:
        raise ValueError("File too large")

    file_hash = hashlib.sha256(content).hexdigest()[:16]
    ext = Path(file.filename).suffix
    filename = f"{file_hash}{ext}"
    filepath = UPLOAD_DIR / filename

    async with aiofiles.open(filepath, 'wb') as f:
        await f.write(content)

    return filename

def process_file(filename: str):
    """Background task to process uploaded file"""
    filepath = UPLOAD_DIR / filename
    # Resize image, extract text from PDF, etc.
    print(f"Processing {filename}")

@app.post("/upload")
async def upload_file(
    file: UploadFile,
    background_tasks: BackgroundTasks
):
    try:
        filename = await save_upload(file)
        background_tasks.add_task(process_file, filename)
        return {"filename": filename, "status": "processing"}
    except ValueError as e:
        raise HTTPException(400, str(e))
```

Using a content hash as the filename provides deduplication and prevents directory traversal attacks. The original filename is never used for storage.

Background tasks keep the upload response fast. For heavy processing, consider a proper task queue like Celery or RQ rather than FastAPI's built-in background tasks.

Chapter 6. Payments

Payment integration requires careful attention to security and error handling. Stripe provides a well-designed API for both one-time payments and subscriptions.

Stripe Checkout Integration

Stripe Checkout handles the payment form, reducing PCI compliance burden:

```
import stripe
from fastapi import FastAPI, Request

stripe.api_key = "sk_test_..."

@app.post("/create-checkout-session")
```

```python
async def create_checkout(price_id: str, user_id: str):
    session = stripe.checkout.Session.create(
        payment_method_types=["card"],
        line_items=[{
            "price": price_id,
            "quantity": 1,
        }],
        mode="subscription",  # or "payment" for one-time
        success_url="https://yoursite.com/success?session_id={CHECKOUT_SESSION_ID}",
        cancel_url="https://yoursite.com/canceled",
        client_reference_id=user_id,
    )
    return {"url": session.url}

@app.post("/webhook")
async def stripe_webhook(request: Request):
    payload = await request.body()
    sig = request.headers.get("stripe-signature")

    try:
        event = stripe.Webhook.construct_event(
            payload, sig, "whsec_..."
        )
    except Exception as e:
        raise HTTPException(400, str(e))

    if event["type"] == "checkout.session.completed":
        session = event["data"]["object"]
        user_id = session["client_reference_id"]
        # Activate subscription for user

    elif event["type"] == "customer.subscription.deleted":
        # Handle cancellation
        pass

    return {"status": "ok"}
```

Always verify webhook signatures. Without verification, anyone could send fake events to your webhook endpoint.

The `client_reference_id` links the Stripe session to your user. This is essential for knowing which account to activate after successful payment.

Part III. DevOps and Infrastructure

Production applications need containerization, automated deployment, monitoring, and database management. These recipes cover the essential patterns.

Chapter 7. Containerization

Docker containers provide consistent environments from development through production.

Multi-Stage Docker Build

Multi-stage builds separate the build environment from the runtime environment, producing smaller, more secure images:

```
# Stage 1: Build
FROM node:20-alpine AS builder
WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build

# Stage 2: Production
FROM node:20-alpine AS production
WORKDIR /app

RUN addgroup -g 1001 -S app && \
    adduser -S -D -H -u 1001 -h /app -s /sbin/nologin -G app app

COPY --from=builder --chown=app:app /app/dist ./dist
COPY --from=builder --chown=app:app /app/node_modules ./node_modules
COPY --from=builder --chown=app:app /app/package.json ./

USER app
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

The build stage includes all development dependencies. The production stage contains only runtime dependencies and compiled code.

Running as a non-root user limits the damage from container escape vulnerabilities. The minimal Alpine base image reduces attack surface.

For Python applications, the pattern is similar:

```
FROM python:3.11-slim AS builder
WORKDIR /app
RUN pip install poetry
COPY pyproject.toml poetry.lock ./
RUN poetry export -f requirements.txt > requirements.txt
RUN pip wheel --no-cache-dir --wheel-dir /wheels -r requirements.txt

FROM python:3.11-slim
WORKDIR /app
COPY --from=builder /wheels /wheels
RUN pip install --no-cache-dir /wheels/*
COPY . .
CMD ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0"]
```

Chapter 8. CI/CD Pipelines

Automated testing and deployment reduces errors and accelerates delivery.

GitHub Actions Workflow

This workflow runs tests on every push, builds a container image, and deploys on merge to main:

```
name: Deploy

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test

      - name: Run linter
        run: npm run lint

  build:
    needs: test
    runs-on: ubuntu-latest
    if: github.event_name == 'push'
    permissions:
      contents: read
      packages: write

    steps:
      - uses: actions/checkout@v4
```

```
      - name: Log in to Container registry
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Build and push
        uses: docker/build-push-action@v5
        with:
          push: true
          tags: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{ github.sha }}

  deploy:
    needs: build
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
      - name: Deploy to production
        run: |
          curl -X POST ${{ secrets.DEPLOY_WEBHOOK }} \
            -H "Authorization: Bearer ${{ secrets.DEPLOY_TOKEN }}" \
            -d '{"image": "${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{ github.sha
}}"}'
```

The `needs` clauses ensure stages run in order. Pull requests run tests but skip build and deploy.

Using the git SHA as the image tag creates immutable deployments. You can always identify exactly which commit is running in production.

Chapter 9. Monitoring

You can't fix problems you don't know about. Monitoring provides visibility into application health and performance.

Prometheus Metrics

Expose application metrics in Prometheus format:

```
from prometheus_client import Counter, Histogram, generate_latest
from fastapi import FastAPI, Response
import time

REQUEST_COUNT = Counter(
    'http_requests_total',
    'Total HTTP requests',
    ['method', 'endpoint', 'status']
)

REQUEST_LATENCY = Histogram(
```

```
    'http_request_duration_seconds',
    'HTTP request latency',
    ['method', 'endpoint']
)

@app.middleware("http")
async def metrics_middleware(request, call_next):
    start = time.time()
    response = await call_next(request)
    duration = time.time() - start

    REQUEST_COUNT.labels(
        method=request.method,
        endpoint=request.url.path,
        status=response.status_code
    ).inc()

    REQUEST_LATENCY.labels(
        method=request.method,
        endpoint=request.url.path
    ).observe(duration)

    return response

@app.get("/metrics")
async def metrics():
    return Response(
        generate_latest(),
        media_type="text/plain"
    )
```

Prometheus scrapes the `/metrics` endpoint periodically. Configure scraping in `prometheus.yml`:

```
scrape_configs:
  - job_name: 'app'
    static_configs:
      - targets: ['app:8000']
    metrics_path: '/metrics'
```

Use Grafana to visualize metrics and create alerts. The combination of Prometheus for collection and Grafana for visualization is the standard observability stack.

Chapter 10. Database Management

Schema changes are inevitable. Managing them safely requires version-controlled migrations.

Alembic Migrations

Alembic tracks schema changes as versioned migration scripts:

```
"""create users table

Revision ID: 001
Create Date: 2024-01-15
"""
from alembic import op
import sqlalchemy as sa

revision = '001'
down_revision = None

def upgrade():
    op.create_table(
        'users',
        sa.Column('id', sa.UUID(), primary_key=True),
        sa.Column('email', sa.String(255), unique=True, nullable=False),
        sa.Column('name', sa.String(255)),
        sa.Column('created_at', sa.DateTime(), server_default=sa.func.now()),
    )
    op.create_index('ix_users_email', 'users', ['email'])

def downgrade():
    op.drop_index('ix_users_email')
    op.drop_table('users')
```

Run migrations with the Alembic CLI:

```
# Create new migration
alembic revision -m "add posts table"

# Run all pending migrations
alembic upgrade head

# Rollback one migration
alembic downgrade -1

# Show current version
alembic current
```

Every migration has both `upgrade` and `downgrade` functions. Test downgrades in development because you may need them in production emergencies.

For TypeScript projects, Prisma provides a more integrated experience:

```
model User {
  id        String   @id @default(uuid())
  email     String   @unique
  name      String?
  posts     Post[]
  createdAt DateTime @default(now())
```

```
}

model Post {
  id        String   @id @default(uuid())
  title     String
  content   String
  author    User     @relation(fields: [authorId], references: [id])
  authorId  String
}
```

```
npx prisma migrate dev --name add_posts
npx prisma migrate deploy
```

Prisma generates migrations automatically from schema changes, reducing the manual work required.

Purple Squirrel Media