The Ultimate Developer Prompt Pack

200+ Ready-to-Use Prompts for Coding, APIs, DevOps & More

By DevForge | Version 1.0 | February 2026

What you get:

- **200+ battle-tested prompts** across 10 essential developer categories
- **Copy-paste ready** — works with ChatGPT, Claude, Gemini, and any LLM
- **Customizable** — every prompt has `[BRACKETED]` placeholders for your specific needs
- **Pro tips included** — each prompt comes with customization advice and expected output descriptions

Categories covered: Code Generation | API Development | DevOps & Infrastructure | Database Design | Debugging & Code Review | Testing | Security | Documentation | Refactoring & Performance | System Design & Architecture

> ***How to use this pack:*** *Each prompt is designed to be copied directly into ChatGPT, Claude, or any LLM. Replace the* `[BRACKETED]` *placeholders with your specific details. Read the "Tips for Customization" to get even better results.*

Table of Contents

---

1. Code Generation Prompts

> *Prompts for generating production-quality code across Python, JavaScript, TypeScript, Go, and Rust. Each prompt enforces best practices, type safety, error handling, and documentation.*

1.1 Python: Production-Ready CLI Tool

Title

Generate a Full CLI Application with Click

Prompt

```
You are a senior Python developer. Build a complete CLI tool using the
Click library that does the following:

Purpose: [DESCRIBE YOUR CLI TOOL'S PURPOSE, e.g., "Batch rename files in
a directory based on regex patterns"]

Requirements:
- Use Click for argument parsing with proper help text for every command
and option
- Implement at least these subcommands: [LIST YOUR SUBCOMMANDS, e.g.,
"preview, execute, undo"]
- Add --verbose and --dry-run flags as global options
- Include proper error handling with user-friendly error messages (no
raw tracebacks)
- Add colored output using Click's style utilities
- Implement a configuration file loader (YAML or TOML) for default
settings
- Add logging with configurable log levels
- Include a progress bar for long-running operations using Click's
progressbar
```

```
- Write complete docstrings for all functions
- Add type hints throughout (compatible with Python 3.9+)

Structure the code as a proper Python package with:
- __main__.py for entry point
- cli.py for Click commands
- core.py for business logic (separated from CLI layer)
- config.py for configuration management
- exceptions.py for custom exceptions

Include a pyproject.toml with the console_scripts entry point
configured.

Do not use any deprecated APIs. Follow PEP 8 and PEP 257.
```

Expected Output

A complete Python package with 5-7 files, fully typed, with Click decorators, subcommands, error handling, and a pyproject.toml. The code will be production-ready and installable via pip.

Tips for Customization

- Replace the purpose and subcommands with your actual needs
- Add `"Include unit tests using pytest with Click's CliRunner"` if you want tests generated too
- Specify `"Use Rich library instead of Click's built-in styling"` for more advanced terminal output
- Add `"Include a Dockerfile for containerized distribution"` for Docker packaging

1.2 Python: Data Processing Pipeline

Title

Generate a Robust Data Pipeline with Validation

Prompt

```
You are a senior data engineer. Build a Python data processing pipeline
with the following specifications:

Data source: [DESCRIBE YOUR INPUT, e.g., "CSV files containing e-
commerce transaction records with columns: transaction_id, user_id,
product_id, amount, currency, timestamp, status"]

Pipeline stages:
1. Ingestion: Read from [SOURCE TYPE, e.g., "CSV files in a directory,
S3 bucket, or database"]
2. Validation: Validate every record using Pydantic v2 models with
strict type checking
3. Transformation: [DESCRIBE TRANSFORMATIONS, e.g., "Convert currencies
to USD, aggregate daily totals, calculate running averages"]
4. Enrichment: [DESCRIBE ENRICHMENT, e.g., "Look up product categories,
add user segments"]
5. Output: Write to [DESTINATION, e.g., "Parquet files partitioned by
date, PostgreSQL table"]

Technical requirements:
- Use Pydantic v2 for all data models with custom validators
- Implement the pipeline using generator functions for memory efficiency
- Add comprehensive error handling: bad records should be logged to a
dead-letter file, not crash the pipeline
- Include retry logic with exponential backoff for external service
calls
- Add pipeline metrics: records processed, records failed, processing
time per stage
- Use structlog for structured logging with JSON output
- Implement graceful shutdown on SIGINT/SIGTERM
- Add a --resume flag that continues from the last successfully
processed record
- Type hint everything

Include:
- models.py (Pydantic models)
- pipeline.py (pipeline orchestration)
- stages/ directory with one file per stage
- metrics.py (pipeline metrics collection)
- main.py (entry point with argument parsing)

Generate the complete code, not pseudocode.
```

Expected Output

A multi-file Python project implementing a full ETL pipeline with Pydantic validation, generator-based processing, error handling, metrics, and structured logging. Each stage is modular and testable.

Tips for Customization

- Be very specific about your data schema for more accurate Pydantic models
- Add `"Use Apache Beam / Prefect / Dagster for orchestration"` if you need a specific framework
- Include sample data in your prompt for even more tailored output
- Add `"Include a Docker Compose setup with a local PostgreSQL for testing"` for infrastructure

1.3 Python: Async Web Scraper with Rate Limiting

Title

Generate a Production Async Scraper with Anti-Ban Features

Prompt

```
You are an expert Python developer specializing in web scraping. Build
an async web scraper with the following specifications:

Target: [DESCRIBE WHAT YOU'RE SCRAPING, e.g., "Product listings from an
e-commerce site with pagination, including product name, price,
description, images, ratings, and review count"]

Technical requirements:
- Use aiohttp for async HTTP requests with connection pooling
- Use BeautifulSoup4 or lxml for HTML parsing (choose the faster option)
- Implement these anti-detection measures:
  * Rotating User-Agent strings (include a realistic list of 20+)
  * Random delays between requests (configurable min/max)
  * Respect robots.txt
  * Session management with cookie persistence
  * Optional proxy rotation support (accept a proxy list file)
- Rate limiting: maximum [N] requests per second with a token bucket
```

```
algorithm
- Implement a URL frontier with BFS crawling strategy and URL
deduplication
- Save results incrementally to [FORMAT, e.g., "JSON Lines file"] so no
data is lost on crash
- Add a checkpoint/resume system using a SQLite database to track
visited URLs
- Implement exponential backoff on 429/503 responses
- Use semaphores to limit concurrent connections
- Add structured logging showing progress (URLs queued, processed,
failed)
- Handle these edge cases:
  * Infinite redirect loops
  * Malformed HTML
  * Connection timeouts
  * Character encoding detection

Structure:
- scraper.py (main scraper class)
- parser.py (HTML parsing and data extraction)
- rate_limiter.py (token bucket implementation)
- models.py (data models for scraped items)
- storage.py (output writers)
- config.py (configuration with sensible defaults)
- main.py (CLI entry point)

All code must use async/await properly. No blocking calls in the async
event loop.
```

Expected Output

A complete async web scraping framework with rate limiting, anti-detection, checkpointing, and modular architecture. Typically 400-600 lines of well-structured, production-quality code.

Tips for Customization

- Describe your target site's structure for more accurate parsers
- Add `"Include Playwright/Selenium fallback for JavaScript-rendered pages"` for dynamic sites
- Add `"Output to both JSON Lines and a PostgreSQL database"` for dual output
- Specify `"Add a monitoring dashboard using Rich Live display"` for visual progress

1.4 JavaScript: Full-Stack React Component Library

Title

Generate a Reusable React Component with Full Features

Prompt

```
You are a senior React developer. Build a production-ready React
component with the following specifications:

Component: [DESCRIBE YOUR COMPONENT, e.g., "A data table with sorting,
filtering, pagination, row selection, column resizing, and CSV export"]

Requirements:
- Use React 18+ with functional components and hooks only
- Write in TypeScript with strict mode enabled
- Implement these features:
  * [FEATURE 1, e.g., "Multi-column sorting with sort indicators"]
  * [FEATURE 2, e.g., "Debounced search/filter with highlight matching"]
  * [FEATURE 3, e.g., "Server-side pagination with configurable page
sizes"]
  * [FEATURE 4, e.g., "Checkbox row selection with select-all"]
  * [FEATURE 5, e.g., "Column resizing via drag handles"]
  * [FEATURE 6, e.g., "CSV/JSON export of selected or filtered data"]
- Props interface:
  * Accept data as a generic typed array
  * Accept column definitions with typed accessors
  * All event callbacks (onSort, onFilter, onPageChange,
onSelectionChange, onExport)
  * Customization slots (renderCell, renderHeader, renderEmpty,
renderLoading)
  * Controlled and uncontrolled modes for all state
- Performance:
  * Virtualized rendering for 10,000+ rows using react-window or similar
  * Memoize expensive computations with useMemo
  * Use React.memo on cell renderers
  * Debounce filter inputs
- Accessibility:
  * Full keyboard navigation (arrow keys, Enter, Space, Tab)
  * ARIA attributes (role, aria-sort, aria-selected, aria-label)
  * Screen reader announcements for sort and filter changes
  * Focus management
- Styling:
```

```
  * Use CSS modules or styled-components (specify which you prefer)
  * Support dark/light theme via CSS custom properties
  * Responsive design with horizontal scroll on mobile
  * No hardcoded colors - everything via theme tokens

Deliver:
1. The main component file
2. TypeScript type definitions (exported)
3. Custom hooks extracted for reusable logic
4. A CSS module or styled-components file
5. A Storybook story file showing all variants
6. A usage example with sample data
```

Expected Output

A fully typed React component with hooks, accessibility, virtualization, Storybook stories, and theme support. Typically 500-800 lines across all files.

Tips for Customization

- Be specific about features you actually need to avoid bloated output
- Add `"Use Tailwind CSS instead of CSS modules"` if that is your stack
- Add `"Include React Testing Library tests for all interactive features"` for tests
- Specify `"Compatible with Next.js App Router (no useLayoutEffect, proper SSR)"` for Next.js projects

1.5 TypeScript: Type-Safe Event System

Title

Generate a Strongly-Typed Publish/Subscribe Event Bus

Prompt

```
You are a TypeScript expert. Build a type-safe event bus / pub-sub
system with the following specifications:

The event bus must enforce type safety at compile time so that:
```

```
1. Event names are restricted to a predefined set (no arbitrary strings)
2. Event handlers receive the correct payload type for their event
3. Emitting an event requires the correct payload type
4. TypeScript shows errors at compile time for type mismatches

Features:
- Define events using a type map interface: { eventName: PayloadType }
- .on(event, handler) - subscribe with typed handler
- .once(event, handler) - subscribe for one emission only
- .off(event, handler) - unsubscribe specific handler
- .off(event) - unsubscribe all handlers for an event
- .emit(event, payload) - emit with type-checked payload
- .waitFor(event, timeout?) - returns Promise<PayloadType> for async
usage
- Wildcard handler .onAny(handler) that receives event name + payload
- Middleware support: .use(middleware) that can transform/block events
before delivery
- Event history: .getLastEvent(eventName) returns the most recent
payload
- Namespace support: bus.namespace('user') returns a scoped bus for
'user.*' events
- Debug mode: when enabled, logs all events to console with timestamps
- Memory management: warn when handler count exceeds a configurable
threshold (like Node.js EventEmitter)

Advanced TypeScript requirements:
- Use conditional types and mapped types where appropriate
- The namespace feature should preserve type safety for scoped events
- All public methods must have complete JSDoc documentation
- Export type utilities: EventNames<Bus>, EventPayload<Bus, EventName>,
EventHandler<Bus, EventName>
- Ensure no 'any' types leak into the public API

Deliver:
1. event-bus.ts - the main implementation
2. types.ts - all type definitions and utilities
3. middleware.ts - built-in middleware (logger, validator, debounce)
4. example.ts - comprehensive usage examples showing type safety
5. Make it work with both ESM and CommonJS
```

Expected Output

A complete type-safe event bus library with advanced TypeScript generics, middleware support, namespaces, and comprehensive type utilities. Demonstrates deep TypeScript knowledge.

Tips for Customization

- Add your actual event map as an example for more relevant output
- Add `"Include Vitest tests that verify both runtime behavior and compile-time type errors"` for testing
- Specify `"Make it framework-agnostic but include a React hook useEvent(bus, eventName)"` for React integration
- Add `"Benchmark against EventEmitter3 and mitt"` for performance comparison

1.6 TypeScript: Zod Schema Generator from Existing Types

Title

Generate Runtime Validators from TypeScript Interfaces

Prompt

````
You are a TypeScript expert specializing in runtime type safety. I have
the following TypeScript interfaces/types that currently only provide
compile-time safety:

```typescript
[PASTE YOUR TYPESCRIPT INTERFACES HERE, e.g.:
interface User {
  id: string;
  email: string;
  name: string;
  role: 'admin' | 'user' | 'moderator';
  preferences: {
    theme: 'light' | 'dark';
    notifications: boolean;
    language: string;
  };
  createdAt: Date;
  metadata?: Record<string, unknown>;
}

interface CreateUserRequest {
  email: string;
  name: string;
````
````

```
  role?: 'admin' | 'user' | 'moderator';
  password: string;
  confirmPassword: string;
}
]
```

For each interface, generate:

1. A Zod schema that exactly mirrors the TypeScript type, including:
    - Appropriate string validations (email, URL, UUID, min/max length)
    - Number validations (min, max, integer, positive)
    - Date validations (past, future, range)
    - Enum validations for union types
    - Custom refinements for business logic (e.g., password === confirmPassword)
    - .transform() where appropriate (e.g., trim strings, normalize emails)
    - Descriptive error messages for every validation rule
2. The inferred TypeScript type from the Zod schema (using z.infer)
3. A "partial" schema variant for PATCH/update operations
4. A "create" schema variant that omits auto-generated fields (id, createdAt)
5. Type-safe parse functions: parseUser(data: unknown): User
6. An error formatter that produces user-friendly validation error messages

Also generate:

- A validation middleware function for Express.js that uses these schemas
- A helper function that converts Zod schemas to JSON Schema (for OpenAPI docs)
- Example usage showing how to validate API request bodies

Ensure the Zod schemas and TypeScript interfaces remain perfectly in sync.

```
### Expected Output
Complete Zod schemas mirroring your TypeScript types with rich
validations, inferred types, variant schemas, parse functions, Express
middleware, and JSON Schema conversion. Typically 200-400 lines.

### Tips for Customization
- Paste your actual interfaces for production-ready output
- Add `"Include schemas for all nested objects as separate reusable
schemas"` for deeply nested types
```

```
- Add `"Generate a form validation hook using react-hook-form +
@hookform/resolvers/zod"` for frontend use
- Specify `"Use Valibot instead of Zod for smaller bundle size"` if you
prefer Valibot

---

## 1.7 Go: Concurrent Worker Pool

### Title
**Generate a Production Go Worker Pool with Graceful Shutdown**

### Prompt
```

You are a senior Go developer. Build a concurrent worker pool implementation with the following specifications:

Use case: [DESCRIBE YOUR USE CASE, e.g., "Process incoming webhook events from a message queue, where each event requires an HTTP call to an external API and a database write"]

Requirements:

- Generic worker pool using Go 1.21+ generics: Pool[T any, R any] where T is input and R is result
- Configurable number of workers (adjustable at runtime via pool.Resize(n))
- Buffered job channel with configurable capacity
- Results channel for collecting outputs
- Graceful shutdown:
  - pool.Shutdown(ctx) - stop accepting new jobs, finish in-progress work, respect context deadline
  - pool.ForceShutdown() - cancel all in-progress work immediately
- Error handling:
  - Per-job error capture (never panic, always recover)
  - Configurable retry with exponential backoff and jitter
  - Dead-letter queue for permanently failed jobs
  - Circuit breaker pattern: stop processing if error rate exceeds threshold

- Observability:
  - Expose metrics: active workers, queued jobs, completed jobs, failed jobs, avg processing time
  - Health check function
  - Structured logging with slog
- Rate limiting: optional token bucket rate limiter on job submission
- Priority queue: support job priorities (high, medium, low)
- Middleware/hooks: OnJobStart, OnJobComplete, OnJobError callbacks
- Context propagation: each job carries its own context for tracing

Structure:

- pool.go (main pool implementation)
- worker.go (worker goroutine logic)
- options.go (functional options pattern for configuration)
- metrics.go (metrics collection)
- circuit_breaker.go (circuit breaker implementation)
- retry.go (retry logic)
- pool_test.go (comprehensive tests including race condition testing)

Use functional options pattern for pool configuration. All public types and functions must have godoc comments. Use the standard library only (no external dependencies) except for testing.

```
### Expected Output
A complete Go worker pool package using generics, channels, graceful
shutdown, circuit breaker, retry logic, and comprehensive tests.
Typically 600-900 lines of idiomatic Go.

### Tips for Customization
- Describe your actual job type for more tailored implementation
- Add `"Include a benchmark test comparing different worker counts"` for
performance testing
- Add `"Make it compatible with OpenTelemetry tracing"` for distributed
tracing
- Specify `"Add a Redis-backed job queue option"` for persistent jobs

---

## 1.8 Go: HTTP Client with Resilience Patterns
```

```
### Title
**Generate a Production HTTP Client Wrapper in Go**

### Prompt
```

You are a senior Go developer. Build a resilient HTTP client wrapper with the following features:

Purpose: A drop-in replacement for http.Client that adds production resilience patterns.

Features:

1. Automatic retries with configurable policy:
    - Retry on 429, 500, 502, 503, 504
    - Exponential backoff with jitter
    - Configurable max retries
    - Respect Retry-After header
    - Custom retry predicate function
1. Circuit breaker (implement from scratch, no external libs):
    - States: Closed, Open, Half-Open
    - Configurable failure threshold, success threshold, timeout duration
    - Per-host circuit breakers
    - State change callbacks
2. Request/Response middleware pipeline:
    - Before-request hooks (add headers, log, trace)
    - After-response hooks (log, metrics, transform)
    - Built-in middleware: logging, metrics, auth token injection, request ID
3. Connection pooling:
    - Per-host connection limits
    - Idle connection timeout
    - DNS caching with TTL
4. Timeout management:
    - Per-request timeout

    - Overall timeout (including retries)
    - Connect timeout vs read timeout

5. Response handling utilities:
    - Generic JSON decoder: DecodeResponseT any (T, error)
    - Response caching with TTL (in-memory LRU cache)
    - Automatic gzip decompression
    - Response body size limit

6. Observability:
    - Structured logging with slog
    - Metrics: request count, latency histogram, error rate (by host, method, status)
    - Request/response dumping in debug mode
    - OpenTelemetry-compatible trace context propagation

API design:

- Use functional options: NewClient(opts ...Option)
- Fluent request builder: client.Get(url).Header(k,v).Query(k,v).Do(ctx)
- Must implement http.RoundTripper interface for compatibility
- Thread-safe for concurrent use

Deliver:

- client.go, options.go, retry.go, circuit_breaker.go, middleware.go, cache.go
- client_test.go with httptest-based tests
- example_test.go with usage examples

Standard library only. Go 1.21+.

```
### Expected Output
A feature-rich, production-grade HTTP client library in Go with retry,
circuit breaker, middleware, caching, and fluent API. Typically 800-1200
lines.

### Tips for Customization
- Add `"Include OAuth2 token management with automatic refresh"` for API
integrations
- Add `"Include Prometheus metrics exporter"` for monitoring
```

```
- Specify `"Optimize for high-throughput scenarios (100k+ requests/
minute)"` for performance tuning
- Add `"Include a mock transport for testing"` for better test utilities

---

## 1.9 Rust: Error Handling Library

### Title
**Generate Comprehensive Rust Error Types with thiserror**

### Prompt
```

You are an expert Rust developer. I'm building a [TYPE OF APPLICATION, e.g., "REST API server"] and need a comprehensive error handling setup.

My application has these layers:

- [LAYER 1, e.g., "HTTP handler layer (Axum)"]
- [LAYER 2, e.g., "Service/business logic layer"]
- [LAYER 3, e.g., "Repository/database layer (SQLx)"]
- [LAYER 4, e.g., "External API client layer (reqwest)"]

For each layer, generate:

1. A dedicated error enum using thiserror with:
    - Variants for every common failure mode in that layer
    - Proper #[from] attributes for automatic conversion from lower layers
    - #[source] annotations for error chaining
    - Human-readable Display messages
    - Structured context (not just strings) on each variant
1. Error conversion implementations:
    - From for UpperLayerError (automatic propagation with ?)
    - Into HTTP response for the handler layer errors:
        - Map to appropriate status codes (400, 401, 403, 404, 409, 422, 500, 503)
        - JSON error response body: { "error": { "code": "...", "message": "...", "details": {...} } }
        - Never leak internal errors to the client (log them, return generic 500)
    - Into problem+json (RFC 7807) format

2. Error context helpers:
    - Extension trait on Result to add context: .context("while doing X")?
    - Typed context: .with_context(|| ErrorContext { operation: "...", resource_id: "..." })?
3. Logging integration:
    - Implement a trait that determines log level based on error type (4xx = warn, 5xx = error)
    - Include tracing span information in errors
    - Redact sensitive fields in error output
4. Testing utilities:
    - assert_error_variant! macro for matching specific error types in tests
    - Error factory functions for test setup

Deliver:

- error/mod.rs (re-exports)
- error/handler.rs (HTTP layer errors)
- error/service.rs (business logic errors)
- error/repository.rs (database errors)
- error/client.rs (external API errors)
- error/context.rs (context extension traits)
- error/response.rs (HTTP response conversion)

Show a complete example of error propagation from database through service to HTTP response.

```
### Expected Output
A complete Rust error handling architecture with thiserror enums,
automatic conversions, HTTP response mapping, context helpers, and
testing utilities. Typically 400-700 lines of idiomatic Rust.

### Tips for Customization
- List your actual application layers for tailored output
- Add `"Include anyhow integration for ad-hoc errors in CLI/scripts"`
for CLI tools
- Specify your web framework (Axum, Actix-web, Rocket) for framework-
specific response types
- Add `"Include error catalog with unique error codes for each variant"`
```

```
for API documentation

---

## 1.10 Rust: Async Stream Processor

### Title
**Generate a Tokio-Based Stream Processing Pipeline in Rust**

### Prompt
```

You are an expert Rust developer with deep async experience. Build an async stream processing pipeline with:

Use case: [DESCRIBE YOUR USE CASE, e.g., "Process a stream of log events from Kafka, parse them, enrich with geo-IP data, and write to ClickHouse"]

Requirements:

- Use Tokio 1.x runtime with tokio-stream
- Generic pipeline stages: Stage trait
- Pipeline builder with type-safe chaining: Pipeline::new().then(parse).then(enrich).then(write)
- Each stage runs in its own Tokio task for true parallelism
- Buffered channels between stages (configurable capacity)
- Backpressure: slow consumers cause producers to wait (bounded channels)
- Fan-out: one stage can feed multiple downstream stages
- Fan-in: multiple upstream stages can merge into one
- Batch processing: accumulate N items or wait T duration, whichever comes first
- Error handling:
  - Per-item errors don't stop the pipeline
  - Error items are routed to a dead-letter channel
  - Stage panics are caught and logged
- Graceful shutdown via CancellationToken
- Metrics per stage: throughput, latency p50/p95/p99, error count, backpressure events
- Dynamic scaling: adjust worker count per stage based on queue depth

Trait definitions:

```
#[async_trait]
trait Stage: Send + Sync + 'static {
    type Input: Send;
    type Output: Send;
    async fn process(&self, input: Self::Input) -> Result<Self::Output,
StageError>;
}
```

Deliver:

- pipeline/mod.rs
- pipeline/stage.rs (Stage trait and built-in stages: Map, Filter, FlatMap, Batch, RateLimit)
- pipeline/builder.rs (type-safe builder)
- pipeline/channel.rs (buffered channels with metrics)
- pipeline/metrics.rs
- pipeline/error.rs
- example.rs showing a 4-stage pipeline

Use only Tokio ecosystem crates. All unsafe must be justified with safety comments.

```
### Expected Output
A complete async stream processing framework in Rust with type-safe
pipeline composition, backpressure, metrics, and graceful shutdown.
Typically 600-1000 lines of advanced Rust.

### Tips for Customization
- Describe your actual data types for concrete implementations
- Add `"Include a Kafka source stage using rdkafka"` for Kafka
integration
- Add `"Include benchmarks using Criterion"` for performance testing
- Specify `"Minimize allocations with zero-copy parsing where possible"`
for high-performance scenarios

---

## 1.11 JavaScript: Node.js Streaming File Processor

### Title
**Generate a Memory-Efficient File Processing Pipeline in Node.js**
```

```
### Prompt
```

You are a senior Node.js developer. Build a streaming file processor that handles files of any size without loading them entirely into memory.

Use case: [DESCRIBE YOUR USE CASE, e.g., "Process multi-gigabyte CSV files: parse rows, validate data, transform fields, and write to multiple outputs (clean data file + error report)"]

Requirements:

- Use Node.js Streams API (Readable, Transform, Writable) properly
- Pipeline composition using stream.pipeline() with proper error propagation
- Implement these transform streams as reusable classes:
  1. CSV/JSON Line Parser (handle quoted fields, escaped characters, malformed lines)
  2. Validator (configurable validation rules per field)
  3. Transformer (configurable field mappings and transformations)
  4. Deduplicator (detect and remove duplicate records using a bloom filter for memory efficiency)
  5. Splitter (route records to different output streams based on conditions)
  6. Batcher (group N records for bulk operations)
  7. Rate Limiter (limit throughput to N records/second)
- Error handling:
  - Bad records are diverted to an error stream (not dropped silently)
  - Each error record includes: original data, error message, line number
  - Stream errors trigger proper cleanup of all file handles
  - Implement highWaterMark tuning
- Performance:
  - Use worker_threads for CPU-intensive transforms
  - Implement backpressure correctly (respect writable.write() return value)
  - Add progress reporting without impacting throughput
  - Memory usage stays constant regardless of file size
- Output formats:
  - CSV with configurable delimiter/quoting

    - JSON Lines (newline-delimited JSON)
    - Parquet (using parquet-wasm or similar)
- Observability:
    - Records processed/second
    - Memory usage
    - Records per output stream
    - Estimated time remaining

Deliver as an ES module package with:

- src/streams/ directory with one file per transform
- src/pipeline.js (pipeline builder)
- src/progress.js (progress reporter)
- src/index.js (public API)
- examples/process-csv.js (example usage)
- TypeScript declarations (.d.ts files)

```
### Expected Output
A complete Node.js streaming file processing toolkit with custom
transform streams, pipeline composition, progress reporting, and
TypeScript declarations. Memory-efficient for multi-GB files.

### Tips for Customization
- Specify your actual file format and fields for tailored parsers
- Add `"Include a web UI showing processing progress using Server-Sent
Events"` for monitoring
- Add `"Support reading from and writing to S3 using @aws-sdk/client-
s3"` for cloud workflows
- Specify `"Add GZIP compression/decompression as a transparent pipeline
stage"` for compressed files

---

## 1.12 JavaScript: Express.js Middleware Collection

### Title
**Generate Production Middleware for Express.js**

### Prompt
```

You are a senior Node.js developer. Build a collection of production-ready Express.js middleware functions that I can drop into any Express app.

Generate these middleware (each as a separate, configurable function):

1. **Request Logger**
    - Structured JSON logging (compatible with ELK/Datadog)
    - Log: method, URL, status code, response time, request ID, user agent, IP
    - Configurable log levels based on status code ranges
    - Redact sensitive headers (Authorization, Cookie) and body fields (password, token)
    - Correlation ID propagation (X-Request-ID header)
2. **Rate Limiter**
    - Sliding window algorithm (not fixed window)
    - Per-route and per-user rate limits
    - Configurable store (in-memory Map with TTL, or Redis interface)
    - Return standard rate limit headers (X-RateLimit-Limit, Remaining, Reset)
    - Custom key extractor (IP, API key, user ID)
    - Whitelist/blacklist support
3. **Request Validator**
    - Validate body, query, params, headers against Joi/Zod schemas
    - Return structured validation errors with field paths
    - Strip unknown fields option
    - Coerce types (string "123" to number 123 for query params)
4. **Error Handler**
    - Catch all errors including async route handler rejections
    - Map known error types to HTTP status codes
    - Structured error response format
    - Development mode: include stack traces
    - Production mode: generic messages, log full details server-side
    - Handle these specifically: ValidationError, AuthenticationError, AuthorizationError, NotFoundError, ConflictError, RateLimitError

5. **Security Headers**
    - Set all OWASP recommended headers
    - Content-Security-Policy with configurable directives
    - Strict-Transport-Security
    - X-Content-Type-Options, X-Frame-Options, X-XSS-Protection
    - Permissions-Policy
    - Configurable CORS with credentials support
6. **Response Caching**
    - In-memory LRU cache with configurable max size
    - Cache key based on: method + URL + relevant headers (Accept, Authorization)
    - Configurable TTL per route
    - Cache invalidation via tags
    - Return X-Cache: HIT/MISS header
    - Skip caching for authenticated requests (configurable)
7. **Request Timeout**
    - Per-route configurable timeouts
    - Send 408 or 504 on timeout
    - Cancel downstream operations (abort controller)
    - Log slow requests above a warning threshold

Each middleware must:

- Use the factory pattern: createMiddleware(options) returns middleware function
- Have full JSDoc documentation with @param, @returns, @example
- Have TypeScript type declarations
- Be independently usable (no dependencies between middleware)
- Include a usage example

Deliver as:

- middleware/ directory with one file per middleware
- middleware/index.js exporting everything
- types/ directory with .d.ts files
- examples/full-app.js showing all middleware composed together

```
### Expected Output
7 production-ready Express middleware functions with configuration,
TypeScript types, and a complete example app showing them all working
together. Typically 800-1200 lines.

### Tips for Customization
- Remove middleware you do not need to reduce output size
- Add `"Use Redis for rate limiter and cache stores"` for distributed
deployments
- Add `"Include Prometheus metrics middleware"` for monitoring
- Specify `"Make compatible with Fastify by using a framework-agnostic
core"` for Fastify support

---

## 1.13 Go: CLI Tool with Cobra and Viper

### Title
**Generate a Feature-Complete CLI Tool in Go**

### Prompt
```

You are a senior Go developer. Build a CLI tool using Cobra and Viper with the following specifications:

Tool purpose: [DESCRIBE YOUR CLI TOOL, e.g., "A Kubernetes namespace manager that can list, create, delete namespaces and manage resource quotas"]

Subcommands: [LIST SUBCOMMANDS, e.g., "list, create, delete, describe, quota set, quota get"]

Requirements:

- Use Cobra for command structure with:
  - Proper Use, Short, Long, Example fields for every command
  - Bash/Zsh/Fish/PowerShell completion generation
  - Input validation in Args and PreRunE
  - Group related commands under parent commands
- Use Viper for configuration with:
  - Config file support (~/.toolname.yaml)

    - Environment variable binding (TOOLNAME_*)
    - Flag -> env var -> config file -> default precedence
    - Config file auto-discovery
- Output formatting:
    - Table format (default for TTY)
    - JSON format (default for pipe)
    - YAML format
    - Auto-detect if stdout is a TTY
    - Colorized output with --no-color flag
    - --output / -o flag to override format
- Interactive mode:
    - Confirmation prompts for destructive operations (--yes to skip)
    - Interactive selection menus using survey/bubbletea
- Error handling:
    - Consistent error format across all commands
    - Exit codes: 0 success, 1 general error, 2 usage error
    - Suggestions for common mistakes ("did you mean ...?")
- Logging:
    - --verbose (-v, -vv, -vvv) for log level control
    - Debug logging to stderr (never stdout)
- Testing:
    - Test helpers for executing commands and capturing output
    - Integration test examples

Project structure: cmd/root.go, cmd/list.go, cmd/create.go, etc. internal/config/ internal/output/ internal/client/ (actual business logic) main.go

Include a Makefile with: build, test, lint, install, completions targets. Include goreleaser.yaml for cross-platform releases.

```
### Expected Output
A complete Go CLI application with Cobra commands, Viper configuration,
multiple output formats, interactive prompts, and release configuration.
Typically 700-1000 lines.
```

```
### Tips for Customization
- Describe your actual subcommands and their flags for tailored output
- Add `"Include a self-update command using go-github"` for auto-update
- Add `"Include a plugin system for extensibility"` for plugin
architecture
- Specify `"Add man page generation"` for Unix distribution

---

## 1.14 Python: FastAPI Background Task System

### Title
**Generate an Async Background Job System with FastAPI**

### Prompt
```

You are a senior Python developer. Build a background task system for a FastAPI application without external dependencies like Celery.

Requirements:

- In-process async task queue using asyncio
- Task definition using decorators:

```
@task(retries=3, timeout=300, priority="high")
async def send_email(to: str, subject: str, body: str) -> bool:
    ...
```

- Task scheduling:
  - Immediate execution: send_email.delay(to="user@example.com", ...)
  - Delayed execution: send_email.delay(..., eta=datetime(2024, 1, 1))
  - Periodic/cron execution: send_email.schedule(cron="0 9 * * MON")
- Task queue backed by:
  - In-memory queue (for development)
  - Redis queue (for production) using redis-py async
  - Abstract interface to add other backends
- Worker pool:
  - Configurable concurrency per priority level

    - Separate queues for high/medium/low priority
    - Worker health monitoring
- Task lifecycle:
    - States: PENDING, RUNNING, SUCCESS, FAILED, RETRYING, CANCELLED
    - State change webhooks/callbacks
    - Task progress reporting (task.update_progress(50, "Halfway done"))
    - Task cancellation via task ID
- Result storage:
    - Store results with configurable TTL
    - Query task status and result by task ID via API endpoint
- Monitoring:
    - FastAPI endpoints: GET /tasks/{id}/status, GET /tasks/stats, POST /tasks/{id}/cancel
    - Queue depth, active workers, success/failure rate metrics
    - Dashboard endpoint returning HTML with task statistics
- Error handling:
    - Automatic retries with exponential backoff
    - Dead letter queue for failed tasks
    - Error notifications (configurable callback)
- Integration:
    - FastAPI lifespan handler to start/stop workers
    - Dependency injection for task queue in route handlers
    - Pydantic models for all API request/response types

Deliver:

- tasks/queue.py (queue abstraction)
- tasks/worker.py (worker pool)
- tasks/decorator.py (task decorator)
- tasks/scheduler.py (periodic task scheduler)
- tasks/storage.py (result storage)
- tasks/models.py (Pydantic models)
- tasks/router.py (FastAPI router for monitoring)
- tasks/__init__.py (public API)

- example_app.py (FastAPI app using the task system)

```
### Expected Output
A complete in-process background task system for FastAPI with
scheduling, retries, monitoring, and Redis support. Typically 600-900
lines of production Python.

### Tips for Customization
- Add `"Use SQLAlchemy async as an alternative result backend"` for
database storage
- Add `"Include task dependency chains:
task_a.then(task_b).then(task_c)"` for workflows
- Specify `"Make tasks serializable for distribution across multiple
processes"` for horizontal scaling
- Add `"Include Prometheus metrics exporter"` for monitoring integration

---

## 1.15 Rust: Configuration Management Library

### Title
**Generate a Layered Configuration System in Rust**

### Prompt
```

You are a senior Rust developer. Build a configuration management library for Rust applications with:

Requirements:

- Type-safe configuration using derive macros:

```
#[derive(Config, Debug, Clone)]
struct AppConfig {
    #[config(env = "APP_HOST", default = "0.0.0.0")]
    host: String,
    #[config(env = "APP_PORT", default = 8080)]
    port: u16,
    #[config(nested)]
    database: DatabaseConfig,
    #[config(secret, env = "APP_API_KEY")]
```

```
    api_key: Secret<String>,
}
```

- Configuration sources with priority (highest to lowest):
  1. Command-line arguments (clap integration)
  2. Environment variables (with configurable prefix)
  3. .env file (dotenv style)
  4. Configuration file (TOML, YAML, JSON - auto-detect by extension)
  5. Default values from the derive attribute
- Features:
  - Nested configuration structs (flattened in env vars: DATABASE_HOST)
  - Secret values: Secret wrapper that redacts in Debug/Display output
  - Validation: #[config(validate = "validate_port_range")] custom validator functions
  - Hot reloading: watch config file for changes, notify via channel
  - Profile support: load base.toml then override with {profile}.toml (dev, staging, prod)
  - Environment-aware: automatically detect profile from APP_ENV
  - Configuration dump: serialize current config to any format (with secrets redacted)
  - Builder pattern for programmatic configuration in tests
- Derive macro implementation:
  - Implement as a proc macro in a separate crate
  - Generate source loading and merging code at compile time
  - Provide clear compile errors for invalid attributes
  - Generate a ConfigMetadata type with field names, types, sources, defaults
- Error handling:
  - Rich error types showing which source failed and why
  - Suggest fixes for common issues (missing required field, invalid type)
  - Collect all errors (don't stop at first) for complete feedback

Deliver:

- config-core/src/lib.rs (traits and types)
- config-derive/src/lib.rs (proc macro)
- config/src/lib.rs (re-exports, integration)

- config/src/sources/ (one file per source: env.rs, file.rs, cli.rs)
- config/src/watcher.rs (hot reload)
- config/tests/ (comprehensive tests)
- example/ (example application)

```
### Expected Output
A complete Rust configuration library with derive macros, multiple
sources, secret handling, hot reload, and validation. Demonstrates
advanced Rust macro programming.

### Tips for Customization
- Specify your actual config structure for concrete examples
- Add `"Include async initialization for configs loaded from remote
sources (Consul, etcd)"` for distributed configs
- Add `"Include a config diff utility for comparing two configs"` for
deployment tooling
- Specify `"Make compatible with #[no_std] for embedded use"` for
embedded Rust

---

## 1.16 Python: Decorator Collection for Production Code

### Title
**Generate a Library of Battle-Tested Python Decorators**

### Prompt
```

You are a senior Python developer. Generate a collection of production-ready decorators that I can drop into any Python project. Each decorator must be:

- Fully typed with generics (using ParamSpec and TypeVar from typing)
- Compatible with both sync and async functions
- Preserving the original function's signature (functools.wraps)
- Tested with at least 2 example use cases

Generate these decorators:

1. @retry(max_attempts=3, backoff=exponential, exceptions=(ConnectionError,), on_retry=callback)
    - Exponential backoff with jitter

- Configurable exception types to retry on
- Optional on_retry callback for logging

2. @cache(ttl=300, maxsize=1000, key=custom_key_fn)
    - TTL-based cache (not just LRU)
    - Thread-safe
    - Custom key function
    - Cache statistics (hits, misses, evictions)
    - .invalidate() method on cached function
3. @timeout(seconds=30, on_timeout=callback)
    - Works with both sync (using threading) and async functions
    - Raises TimeoutError with context
    - Cleanup callback
4. @rate_limit(calls=10, period=60, strategy="sliding_window")
    - Sliding window rate limiting
    - Per-key rate limiting (e.g., per user)
    - Thread-safe
    - Return retry-after hint on limit exceeded
5. @circuit_breaker(failure_threshold=5, recovery_timeout=30, expected_exception=Exception)
    - Three states: closed, open, half-open
    - State change callbacks
    - Expose state via .circuit_state property
6. @validate(input_schema=PydanticModel, output_schema=PydanticModel)
    - Validate function arguments against Pydantic model
    - Validate return value against Pydantic model
    - Clear error messages with field paths
7. @log_execution(level="INFO", log_args=True, log_result=True, max_str_len=200)
    - Log function entry, exit, and exceptions
    - Configurable argument/result logging with truncation

    - Redact sensitive parameters by name (password, token, secret)

8. @deprecated(reason="Use new_function instead", removal_version="2.0.0")
    - Issue DeprecationWarning with helpful message
    - Add deprecation notice to docstring
    - Log deprecation on first call only
9. @singleton
    - Thread-safe singleton pattern
    - Support for classes (not just functions)
    - .reset() method for testing
10. @throttle(min_interval=1.0)
    - Ensure minimum time between calls
    - Queue excess calls (don't drop them)
    - Async-compatible

Deliver:

- decorators.py (all decorators in one file with full docstrings)
- decorators_async.py (async-specific implementations if different)
- examples.py (usage examples for each decorator)
- Type stubs if the typing is too complex for inline

Each decorator must compose well with others: @retry @timeout @log_execution should work correctly stacked.

```
### Expected Output
10 production-ready Python decorators with full typing, sync/async
support, docstrings, and usage examples. Typically 600-900 lines of
heavily-documented code.

### Tips for Customization
- Remove decorators you don't need to get more detailed output on the
ones you do
- Add `"Include a @feature_flag(flag_name, default=False) decorator"`
for feature flags
- Add `"Include a @profile decorator that measures CPU time and memory
allocation"` for profiling
```

```
- Specify `"All decorators must work with Django/Flask view functions"`
for web framework compatibility

---

## 1.17 TypeScript: State Machine with Type Safety

### Title
**Generate a Type-Safe Finite State Machine in TypeScript**

### Prompt
```

You are a TypeScript expert. Build a type-safe finite state machine library where invalid transitions are caught at compile time.

Define a state machine for: [DESCRIBE YOUR USE CASE, e.g., "An order processing system with states: draft, submitted, payment_pending, paid, processing, shipped, delivered, cancelled, refunded"]

Requirements:

- State machine definition as a type-level config:

```
const orderMachine = createMachine({
  initial: 'draft',
  states: {
    draft: { on: { SUBMIT: 'submitted', DELETE: 'cancelled' } },
    submitted: { on: { PAY: 'payment_pending', CANCEL:
'cancelled' } },
    // ... etc
  },
  context: { orderId: '', items: [], total: 0 },
});
```

- Compile-time safety:
  - machine.send('INVALID_EVENT') → TypeScript error
  - machine.send('PAY') when in 'draft' state → TypeScript error
  - Transition actions receive correctly typed context and event
  - Guards and actions are typed per-transition

- Runtime features:
  - Guards: conditional transitions based on context/event
  - Actions: side effects on enter/exit/transition (typed per state)
  - Context: typed mutable data associated with the machine
  - Hierarchical states (nested state machines)
  - History states (remember last child state)
  - Invoke: run async services and transition on resolve/reject
- Persistence:
  - Serialize machine state to JSON
  - Restore machine from serialized state
  - Event sourcing: replay events to rebuild state
- Observability:
  - Subscribe to state changes
  - Subscribe to specific transitions
  - Transition history log
  - Visualize as Mermaid state diagram (generate markdown)
- Integration:
  - React hook: useMachine(definition) returning [state, send, context]
  - Event handler: onTransition, onStateEnter, onStateExit

Deliver:

- machine.ts (core implementation)
- types.ts (all type-level magic)
- hooks.ts (React integration)
- serialization.ts (persistence)
- visualize.ts (Mermaid diagram generator)
- example.ts (complete order processing example)

Minimize use of 'any' and 'as' type assertions. Prefer type inference.

```
### Expected Output
```

```
A complete type-safe state machine library with compile-time transition
validation, React integration, persistence, and visualization.
Demonstrates advanced TypeScript type programming.

### Tips for Customization
- Replace the order example with your actual state machine
- Add `"Include XState v5 compatibility layer"` for XState migration
- Add `"Include a visual state machine editor component"` for tooling
- Specify `"Generate Zod schemas for event validation"` for runtime
safety

---

## 1.18 Python: Custom ORM Query Builder

### Title
**Generate a Type-Safe SQL Query Builder in Python**

### Prompt
```

You are a senior Python developer specializing in database tooling. Build a type-safe SQL query builder that generates parameterized queries.

Requirements:

- Fluent API:

```
query = (
    Query(users)
    .select(users.id, users.name, users.email)
    .join(orders, on=users.id == orders.user_id)
    .where(users.age >= 18, users.status == 'active')
    .where(orders.total > 100)  # AND by default
    .or_where(users.role == 'admin')
    .order_by(users.name.asc(), orders.total.desc())
    .limit(20)
    .offset(40)
    .build()
)
# query.sql = "SELECT users.id, users.name, ... WHERE ... ORDER
BY ... LIMIT $1 OFFSET $2"
# query.params = [18, 'active', 100, 'admin', 20, 40]
```

- Table/Column definitions using descriptors:

```
class Users(Table):
    id = Column(Integer, primary_key=True)
    name = Column(String(100), nullable=False)
    email = Column(String(255), unique=True)
    age = Column(Integer)
    status = Column(String(20), default='active')
```

- Query types: SELECT, INSERT, UPDATE, DELETE, UPSERT
- JOIN types: INNER, LEFT, RIGHT, FULL OUTER, CROSS
- WHERE operators: ==, !=, >, <, >=, <=, IN, NOT IN, LIKE, ILIKE, BETWEEN, IS NULL, IS NOT NULL
- Aggregate functions: COUNT, SUM, AVG, MIN, MAX with GROUP BY and HAVING
- Subquery support: WHERE id IN (SELECT ...)
- Window functions: ROW_NUMBER, RANK, LAG, LEAD with OVER/PARTITION BY
- Common Table Expressions (CTEs): WITH ... AS (...)
- UNION, INTERSECT, EXCEPT
- Raw SQL escape hatch: .raw("custom SQL", params)

Safety:

- All user values are parameterized (never interpolated into SQL)
- SQL injection prevention by design
- Validate column names against table definitions
- Dialect support: PostgreSQL ($1, $2) and MySQL (%s)

Type safety:

- Column comparisons are type-checked (can't compare Integer to String)
- Select result type is inferred from selected columns
- Full type hints throughout

Deliver:

- query_builder/table.py (Table and Column definitions)
- query_builder/query.py (Query builder with all methods)
- query_builder/expressions.py (WHERE conditions and operators)
- query_builder/functions.py (SQL functions: COUNT, SUM, etc.)
- query_builder/dialects.py (PostgreSQL, MySQL parameterization)

- query_builder/types.py (Column types)
- examples.py (15+ example queries showing all features)

```
### Expected Output
A complete SQL query builder with fluent API, type safety, multiple
query types, window functions, CTEs, and dialect support. Typically
500-800 lines.

### Tips for Customization
- Specify your database (PostgreSQL, MySQL, SQLite) for dialect-specific
features
- Add `"Include async execution using asyncpg/aiomysql"` for async
database access
- Add `"Include migration generation from table definitions"` for schema
management
- Specify `"Generate SQLAlchemy Core compatible expressions"` for
SQLAlchemy integration

---

## 1.19 JavaScript: WebSocket Manager with Auto-Reconnect

### Title
**Generate a Robust WebSocket Client with Reconnection Logic**

### Prompt
```

You are a senior JavaScript/TypeScript developer. Build a production WebSocket client manager with the following features:

Requirements:

- TypeScript with strict mode, generic message types
- Auto-reconnection:
  - Exponential backoff with jitter (configurable min/max delay)
  - Maximum reconnection attempts (configurable, default unlimited)
  - Reconnection state machine: CONNECTING, CONNECTED, RECONNECTING, DISCONNECTED, FAILED
  - Resume subscriptions after reconnect
  - Queue messages sent during disconnection, flush on reconnect

    - Detect connection staleness via heartbeat/ping-pong

- Message handling:
    - Type-safe message sending: ws.send({ type: 'chat', text: 'hello' })
    - Type-safe message receiving: ws.on('chat', (msg) => ...)
    - Request/response pattern: const resp = await ws.request('getData', payload, { timeout: 5000 })
    - Message serialization/deserialization (JSON by default, pluggable for msgpack/protobuf)
    - Message deduplication (by message ID)

- Channel/room support:
    - Subscribe to channels: ws.subscribe('room:123')
    - Unsubscribe: ws.unsubscribe('room:123')
    - Channel-scoped message handling
    - Auto-resubscribe after reconnect

- Authentication:
    - Token-based auth on initial connect (sent as first message or query param)
    - Token refresh: callback to get new token before reconnect
    - Handle 401/403 disconnect without reconnecting

- Observability:
    - Connection state change events
    - Message sent/received counters
    - Latency measurement (via heartbeat round-trip)
    - Bandwidth tracking
    - Event log with timestamps

- Browser + Node.js:
    - Use native WebSocket in browser
    - Use 'ws' package in Node.js
    - Isomorphic API
    - Handle page visibility API (reconnect on tab focus)

Deliver:

- src/client.ts (main WebSocket manager)
- src/reconnect.ts (reconnection strategy)
- src/channel.ts (channel/subscription management)
- src/message.ts (message serialization, typing, dedup)
- src/heartbeat.ts (keep-alive logic)
- src/types.ts (all TypeScript types)
- src/index.ts (public API)
- examples/chat-client.ts (example usage)

No external dependencies except 'ws' for Node.js.

```
### Expected Output
A production WebSocket client library with reconnection, typed messages,
channels, authentication, and cross-platform support. Typically 500-800
lines of TypeScript.

### Tips for Customization
- Specify your actual message types for tailored type definitions
- Add `"Include a React hook: useWebSocket(url, options)"` for React
integration
- Add `"Include a Vue composable: useWebSocket(url, options)"` for Vue
integration
- Specify `"Include binary message support using ArrayBuffer"` for
binary protocols

---

## 1.20 Go: Database Migration Tool

### Title
**Generate a SQL Migration System in Go**

### Prompt
```

You are a senior Go developer. Build a database migration tool similar to golang-migrate but simpler and tailored to my needs.

Requirements:

- Migration file format:
  - Naming: YYYYMMDDHHMMSS_description.up.sql and .down.sql
  - Or: Go-based migrations for complex data transformations
  - Automatic ordering by timestamp
- Commands:
  - migrate up [N] - apply N pending migrations (all if N omitted)
  - migrate down [N] - rollback N migrations
  - migrate status - show applied/pending migrations
  - migrate create NAME - generate new migration file pair
  - migrate force VERSION - manually set migration version (for fixing stuck state)
  - migrate validate - check migration files for issues
- Database support:
  - PostgreSQL (primary)
  - MySQL (secondary)
  - Abstract interface for adding others
  - Connection string parsing and validation
- Safety features:
  - Advisory lock to prevent concurrent migrations
  - Transaction per migration (with option to disable for DDL that can't be transactional)
  - Dry-run mode (show SQL without executing)
  - Checksum verification (detect modified applied migrations)
  - Backup reminder before destructive migrations
  - Migration plan preview before execution
- Migration tracking:
  - schema_migrations table with: version, name, applied_at, checksum, execution_time_ms
  - Support for dirty state detection and recovery
- Developer experience:
  - Embeddable as a library (not just CLI)

    - Go API: migrator.Up(ctx, db, migrationsFS)
    - embed.FS support for embedded migrations
    - Configurable logger interface
    - Programmatic migration definitions for complex scenarios
- Extra features:
    - Seed data management (separate from migrations)
    - Schema dump to SQL file
    - Migration squashing (combine old migrations)

Deliver:

- migrate/migrator.go (core logic)
- migrate/source.go (migration source: filesystem, embed.FS)
- migrate/database/ (database drivers)
- migrate/lock.go (advisory locking)
- cmd/migrate/main.go (CLI using cobra)
- migrate_test.go (tests using testcontainers-go for real DB testing)

```
### Expected Output
A complete database migration tool in Go with CLI, library API, multi-database support, safety features, and real database tests. Typically 700-1000 lines.

### Tips for Customization
- Specify your database for focused output
- Add `"Include Atlas-style declarative migrations"` for schema-as-code
- Add `"Include a GitHub Action for running migrations in CI"` for CI/CD
- Specify `"Support multi-tenant migrations (per-tenant schema)"` for SaaS applications

---

## 1.21 TypeScript: Full-Stack Type Sharing

### Title
**Generate a Type-Safe API Layer Shared Between Frontend and Backend**

### Prompt
```

You are a full-stack TypeScript developer. Build a shared type system and API client that provides end-to-end type safety between a [BACKEND FRAMEWORK, e.g., "Express/Fastify"] backend and a [FRONTEND FRAMEWORK, e.g., "React/Next.js"] frontend.

API routes: [LIST YOUR API ROUTES, e.g.:

- GET /api/users - list users with pagination
- GET /api/users/:id - get user by ID
- POST /api/users - create user
- PUT /api/users/:id - update user
- DELETE /api/users/:id - delete user
- GET /api/users/:id/orders - get user's orders]

Requirements:

- Shared package (@myapp/shared):
  - API contract types: route path, method, request body, query params, response body, error responses
  - Zod schemas for runtime validation (shared between client and server)
  - Type inference from Zod schemas (z.infer)
  - Branded types for IDs (UserId, OrderId - not just string)
  - Shared error codes enum
  - API versioning support
- Backend (server/):
  - Route handlers with typed request and response
  - Request validation middleware using shared Zod schemas
  - Response type checking (ensure handlers return correct types)
  - OpenAPI spec generation from the shared types
- Frontend (client/):
  - Type-safe API client: api.users.get(userId) returns Promise
  - Auto-complete for all route parameters, query params, and body fields
  - Error handling with typed error responses
  - React Query/SWR integration with typed hooks: useQuery: const { data } = useUsers() // data is typed as User[] useMutation: const { mutate } = useCreateUser() // mutate accepts CreateUserRequest

    - Optimistic update helpers

- Monorepo setup:
    - Package structure for npm workspaces or turborepo
    - tsconfig paths for cross-package imports
    - Build pipeline that validates types across packages

Deliver:

- packages/shared/src/api.ts (API contract)
- packages/shared/src/schemas.ts (Zod schemas)
- packages/shared/src/types.ts (shared types)
- packages/server/src/routes.ts (typed route handlers)
- packages/server/src/middleware.ts (validation middleware)
- packages/client/src/api-client.ts (typed API client)
- packages/client/src/hooks.ts (React Query hooks)
- tsconfig.json files for each package
- package.json files for each package ```

Expected Output

A complete monorepo setup with shared types, backend route handlers, and a frontend API client, all connected with end-to-end type safety. Typically 500-800 lines across all packages.

Tips for Customization

- Replace the example routes with your actual API
- Add `"Use tRPC instead of manual type sharing"` if you want the tRPC approach
- Add `"Include Tanstack Query v5 integration"` for modern React Query
- Specify `"Use Hono instead of Express for edge runtime compatibility"` for edge deployments

Want all 200+ prompts across 10 categories? **Get the full Developer Prompt Pack for just $19** The full pack includes prompts for: Code Generation, API Development, DevOps, Database Design, Debugging, Testing, Security, Documentation, Refactoring, and System Design. **Visit devforge.tools to get the complete pack!**