Phase 2: Database Enhancements - Implementation Plan

Date: November 15, 2025
Repository: https://github.com/StreallyX/payroll-saas
Branch: dev
Priority: 🟡 HIGH
Duration: 2-3 weeks
Current Progress: 13% completed

📋 Table of Contents

🎯 Overview

Phase 2 focuses on enhancing the database schema to support advanced features required for a Deel-like payroll SaaS platform. This phase includes:

- **Database Schema Improvements:** Add missing fields and optimize existing models
- **New Tables:** Create 24+ new tables for payments, timesheets, expenses, approvals, and more
- **Indexing Strategy:** Add proper indexes for query performance
- **Data Relationships:** Establish proper relationships between Tenant and all entities

Why This Phase Matters:
- Establishes the foundation for advanced workflow features (Phase 7)
- Enables proper tracking of payments, expenses, and time
- Supports document management and custom fields
- Prepares infrastructure for notifications and webhooks

🔍 Current State Analysis

✅ What Already Exists

Based on the analysis of the Prisma schema, the following tables are already implemented:

Category	Table	Status	Notes
Contracts	Contract	✅ Complete	Includes workflow status, termination tracking
	ContractDocument	✅ Complete	Document storage for contracts
	ContractStatusHis-tory	✅ Complete	Tracks contract status changes
	ContractNotification	✅ Complete	Contract-related noti-fications
Invoices	Invoice	✅ Complete	Core invoice model with status tracking
	InvoiceLineItem	✅ Complete	Detailed line items for invoices
Webhooks	WebhookSubscription	✅ Complete	Webhook configura-tion
	WebhookDelivery	✅ Complete	Delivery logs for web-hooks
Notifications	EmailLog	✅ Complete	Email tracking
	SMSLog	✅ Complete	SMS tracking
	NotificationPrefer-ence	✅ Complete	User notification pref-erences
System	ScheduledJob	✅ Complete	Scheduled tasks con-figuration
	SystemConfig	✅ Complete	System-wide config-uration
	Session	✅ Complete	Via NextAuth
	PasswordResetToken	✅ Complete	Password reset func-tionality

❌ What’s Missing

The following tables need to be created as part of Phase 2:

#	Table Name	Priority	Purpose
1	Payment	🔴 Critical	Track all payments (invoices, payroll, expenses)
2	PaymentMethod	🔴 Critical	Store payment method details
3	Expense	🔴 Critical	Track contractor/employee expenses
4	Timesheet	🔴 Critical	Time tracking for contractors
5	TimesheetEntry	🔴 Critical	Individual time entries
6	ApprovalWorkflow	🟡 High	Generic approval workflow tracking
7	ApprovalStep	🟡 High	Individual approval steps
8	Document	🟡 High	Generic document storage (not just contracts)
9	Comment	🟡 High	Comments on any entity
10	Tag	🟢 Medium	Categorization system
11	TagAssignment	🟢 Medium	Tag to entity mapping
12	CustomField	🟢 Medium	Dynamic custom field definitions
13	CustomFieldValue	🟢 Medium	Custom field values per entity
14	UserActivity	🟡 High	Detailed user activity tracking
15	ApiKey	🟡 High	API key management for integrations

🔧 Database Optimizations Needed

Missing Indexes:
- User table: index on `email`, `roleId`, `isActive`
- Contract table: index on `companyId`, `payrollPartnerId`
- Invoice table: index on `invoiceNumber`, `issueDate`, `dueDate`
- Composite indexes for common query patterns

Missing User Fields:
- Profile picture URL
- Phone number
- Timezone
- Language preference
- Last login timestamp
- Email verification status
- Two-factor authentication settings

📝 Phase 2 Requirements

Category 1: Database Schema Improvements (Tasks 51-55)

Task 51: Add Missing Fields to User Model

Current User Model Limitations:
- No profile picture field
- Missing phone number
- No timezone/language preferences
- No last login tracking
- No 2FA settings

Required New Fields:

```
model User {
  // Existing fields...

  // Profile Information
  profilePictureUrl String?
  phone             String?
  timezone          String?  @default("UTC")
  language          String?  @default("en")

  // Authentication & Security
  emailVerified     Boolean  @default(false)
  lastLoginAt       DateTime?
  twoFactorEnabled  Boolean  @default(false)
  twoFactorSecret   String?

  // Preferences
  preferences       Json?    // JSON object for flexible preferences

  // Metadata
  lastActivityAt    DateTime?

  // Relations (new)
  activities        UserActivity[]
  apiKeys           ApiKey[]
  comments          Comment[]
  customFieldValues CustomFieldValue[]
}
```

Task 52: Create Proper Relationships Between Tenant and All Entities

Issue: Some entities don't have direct tenantId relationships
Solution: Ensure all major entities have proper tenant relationships for data isolation

Task 53: Add Indexes on Frequently Queried Fields

Performance Issue: Queries are slow on large datasets
Required Indexes:

```
@@index([tenantId, email])
@@index([tenantId, isActive])
@@index([tenantId, roleId])
@@index([createdAt])
@@index([lastLoginAt])
```

Task 54: Implement Composite Indexes for Complex Queries

Common Query Patterns Needing Optimization:
- Find contracts by tenant + status + date range
- Find invoices by tenant + status + due date
- Find users by tenant + role + active status

Required Composite Indexes:

```
// Contract
@@index([tenantId, status, startDate])
@@index([tenantId, workflowStatus, endDate])

// Invoice
@@index([tenantId, status, dueDate])
@@index([tenantId, status, issueDate])
```

Task 55: Add Unique Constraints Where Necessary

Data Integrity Requirements:

```
// Ensure unique API keys
@@unique([tenantId, key])

// Ensure unique invoice numbers per tenant
@@unique([tenantId, invoiceNumber])
```

Category 2: New Tables to Create (Tasks 56-80)

Task 61: Create Payment Tracking Table

Purpose: Track all payment transactions across the platform

```
model Payment {
  id                String    @id @default(cuid())
  tenantId          String

  // What is being paid
  invoiceId         String?
  expenseId         String?
  payrollRunId      String?   // For future payroll processing

  // Payment details
  amount            Decimal   @db.Decimal(12, 2)
  currency          String
  status            String    // pending, processing, completed, failed, refunded
  paymentMethod     String    // bank_transfer, credit_card, paypal, stripe, etc.
  paymentMethodId   String?   // Reference to PaymentMethod table

  // Transaction details
  transactionId     String?   // External payment gateway transaction ID
  referenceNumber   String?   // Internal reference

  // Dates
  scheduledDate     DateTime?
  processedDate     DateTime?
  completedDate     DateTime?

  // Additional info
  description       String?   @db.Text
  notes             String?   @db.Text
  metadata          Json?     // Flexible data storage

  // Failure handling
  failureReason     String?   @db.Text
  retryCount        Int       @default(0)

  // Audit
  createdById       String
  createdAt         DateTime  @default(now())
  updatedAt         DateTime  @updatedAt

  // Relations
  tenant            Tenant          @relation(fields: [tenantId], references: [id], o
nDelete: Cascade)
  invoice           Invoice?        @relation(fields: [invoiceId], references: [id])
  expense           Expense?        @relation(fields: [expenseId], references: [id])
  paymentMethod     PaymentMethod?  @relation(fields: [paymentMethodId], references:
[id])

  @@index([tenantId])
  @@index([invoiceId])
  @@index([expenseId])
  @@index([status])
  @@index([scheduledDate])
  @@index([createdAt])
  @@map("payments")
}
```

Task 62: Add Payment Method Storage Table

Purpose: Store payment methods for companies, contractors, and agencies

```
enum PaymentMethodType {
  BANK_ACCOUNT
  CREDIT_CARD
  DEBIT_CARD
  PAYPAL
  STRIPE
  WISE
  REVOLUT
  OTHER
}

model PaymentMethod {
  id                String            @id @default(cuid())
  tenantId          String

  // Owner (polymorphic)
  ownerId           String            // User, Company, or Agency ID
  ownerType         String            // "user", "company", "agency"

  // Method details
  type              PaymentMethodType
  isDefault         Boolean           @default(false)

  // Bank account details (if applicable)
  bankName          String?
  accountHolderName String?
  accountNumber     String?           // Encrypted
  routingNumber     String?           // Encrypted
  swiftCode         String?
  iban              String?

  // Card details (if applicable) - Store tokenized version
  cardLast4         String?
  cardBrand         String?           // Visa, Mastercard, etc.
  cardExpMonth      Int?
  cardExpYear       Int?
  cardholderName    String?

  // External gateway reference
  gatewayType       String?           // stripe, paypal, etc.
  gatewayToken      String?           // Token from payment gateway

  // Status
  isActive          Boolean           @default(true)
  isVerified        Boolean           @default(false)
  verifiedAt        DateTime?

  // Audit
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  payments          Payment[]

  @@index([tenantId])
  @@index([ownerId, ownerType])
  @@index([isDefault])
  @@map("payment_methods")
}
```

Task 63: Implement Expense Tracking Table

Purpose: Track expenses submitted by contractors and employees

```
model Expense {
  id                String            @id @default(cuid())
  tenantId          String

  // Who submitted
  submittedById     String
  contractorId      String?
  contractId        String?

  // Expense details
  title             String
  description       String?           @db.Text
  amount            Decimal           @db.Decimal(10, 2)
  currency          String
  category          String            // travel, meals, equipment, software, etc.

  // Receipt/Documentation
  receiptUrl        String?
  receiptFileName   String?

  // Dates
  expenseDate       DateTime
  submittedAt       DateTime          @default(now())

  // Approval workflow
  status            String            // draft, submitted, approved, rejected, paid
  approvalWorkflowId String?
  approvedById      String?
  approvedAt        DateTime?
  rejectionReason   String?           @db.Text

  // Payment tracking
  paymentId         String?
  paidAt            DateTime?

  // Additional info
  notes             String?           @db.Text
  metadata          Json?

  // Audit
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  contractor        Contractor?       @relation(fields: [contractorId], references: [i
d])
  contract          Contract?         @relation(fields: [contractId], references:
[id])
  approvalWorkflow  ApprovalWorkflow? @relation(fields: [approvalWorkflowId], refer-
ences: [id])
  payment           Payment?          @relation(fields: [paymentId], references: [id])
  comments          Comment[]

  @@index([tenantId])
  @@index([submittedById])
  @@index([contractorId])
  @@index([status])
  @@index([expenseDate])
  @@map("expenses")
}
```

Task 64: Create Timesheet Entry Table

Purpose: Track time worked by contractors

```
model Timesheet {
  id                String            @id @default(cuid())
  tenantId          String

  // Who and what
  contractorId      String
  contractId        String?

  // Period
  startDate         DateTime
  endDate           DateTime

  // Status
  status            String            // draft, submitted, approved, rejected, in-
voiced
  approvalWorkflowId String?
  approvedById      String?
  approvedAt        DateTime?
  rejectionReason   String?           @db.Text

  // Totals (calculated from entries)
  totalHours        Decimal           @db.Decimal(10, 2)
  totalAmount       Decimal?          @db.Decimal(10, 2)

  // Invoice linkage
  invoiceId         String?

  // Dates
  submittedAt       DateTime?

  // Audit
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  contractor        Contractor        @relation(fields: [contractorId], references: [i
d])
  contract          Contract?         @relation(fields: [contractId], references:
[id])
  entries           TimesheetEntry[]
  approvalWorkflow  ApprovalWorkflow? @relation(fields: [approvalWorkflowId], refer-
ences: [id])
  invoice           Invoice?          @relation(fields: [invoiceId], references: [id])
  comments          Comment[]

  @@index([tenantId])
  @@index([contractorId])
  @@index([contractId])
  @@index([status])
  @@index([startDate, endDate])
  @@map("timesheets")
}

model TimesheetEntry {
  id                String            @id @default(cuid())
  timesheetId       String

  // Work details
  date              DateTime
  hours             Decimal           @db.Decimal(5, 2)
```

```
  description       String?           @db.Text

  // Optional project/task tracking
  projectName       String?
  taskName          String?

  // Rates (if applicable)
  rate              Decimal?          @db.Decimal(10, 2)
  amount            Decimal?          @db.Decimal(10, 2)

  // Break times
  breakHours        Decimal?          @db.Decimal(5, 2)

  // Audit
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  timesheet         Timesheet         @relation(fields: [timesheetId], references:
[id], onDelete: Cascade)

  @@index([timesheetId])
  @@index([date])
  @@map("timesheet_entries")
}
```

Task 65: Add Approval Workflow Tracking Table

Purpose: Generic approval workflow system for expenses, timesheets, contracts, etc.

```
model ApprovalWorkflow {
  id                String            @id @default(cuid())
  tenantId          String

  // What entity is being approved (polymorphic)
  entityType        String            // expense, timesheet, contract, etc.
  entityId          String

  // Workflow configuration
  workflowType      String            // single_approver, multi_step, parallel, etc.

  // Current status
  status            String            // pending, in_progress, approved, rejected, can
celled
  currentStepOrder  Int               @default(1)

  // Final decision
  finalDecision     String?           // approved, rejected
  finalDecisionAt   DateTime?
  finalDecisionBy   String?

  // Dates
  startedAt         DateTime          @default(now())
  completedAt       DateTime?

  // Metadata
  metadata          Json?

  // Audit
  createdById       String
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  steps             ApprovalStep[]
  expenses          Expense[]
  timesheets        Timesheet[]

  @@index([tenantId])
  @@index([entityType, entityId])
  @@index([status])
  @@map("approval_workflows")
}

model ApprovalStep {
  id                String            @id @default(cuid())
  workflowId        String

  // Step configuration
  stepOrder         Int
  stepName          String

  // Approver
  approverId        String
  approverType      String            // user, role, any_of_role

  // Status
  status            String            // pending, approved, rejected, skipped
  decision          String?           // approve, reject
  decisionAt        DateTime?
```

```
  comments          String?           @db.Text

  // Rules
  isRequired        Boolean           @default(true)

  // Audit
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  workflow          ApprovalWorkflow  @relation(fields: [workflowId], references:
[id], onDelete: Cascade)

  @@index([workflowId])
  @@index([status])
  @@map("approval_steps")
}
```

Task 66-68: Document Management System

Purpose: Generic document storage, comments, and metadata

```
model Document {
  id                String            @id @default(cuid())
  tenantId          String

  // Entity association (polymorphic)
  entityType        String?           // contract, invoice, expense, user, etc.
  entityId          String?

  // File details
  name              String
  description       String?           @db.Text
  fileUrl           String
  fileName          String
  fileSize          Int
  mimeType          String

  // Categorization
  category          String?           // contract, invoice, receipt, id_document, etc.

  // Version control
  version           Int               @default(1)
  isLatestVersion   Boolean           @default(true)
  parentDocumentId  String?

  // Access control
  visibility        String            @default("private") // private, tenant, public

  // Status
  isActive          Boolean           @default(true)

  // Signature tracking
  requiresSignature Boolean           @default(false)
  isSigned          Boolean           @default(false)
  signedAt          DateTime?
  signedBy          String?

  // Upload info
  uploadedById      String
  uploadedAt        DateTime          @default(now())

  // Audit
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  parentDocument    Document?         @relation("DocumentVersions", fields: [parent-
DocumentId], references: [id])
  versions          Document[]        @relation("DocumentVersions")
  comments          Comment[]
  tags              TagAssignment[]

  @@index([tenantId])
  @@index([entityType, entityId])
  @@index([category])
  @@index([uploadedAt])
  @@map("documents")
}

model Comment {
  id                String            @id @default(cuid())
```

```
  tenantId          String

  // Entity association (polymorphic)
  entityType        String            // contract, invoice, expense, timesheet, docu-
ment, etc.
  entityId          String

  // Comment details
  content           String            @db.Text

  // Reply/Thread support
  parentCommentId   String?

  // Status
  isEdited          Boolean           @default(false)
  isDeleted         Boolean           @default(false)

  // Author
  authorId          String
  authorName        String            // Denormalized for deleted users

  // Audit
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  parentComment     Comment?          @relation("CommentReplies", fields: [parentCom-
mentId], references: [id])
  replies           Comment[]         @relation("CommentReplies")

  @@index([tenantId])
  @@index([entityType, entityId])
  @@index([authorId])
  @@index([createdAt])
  @@map("comments")
}
```

Task 69: Tag System for Categorization

Purpose: Flexible tagging system for organizing entities

```
model Tag {
  id                String            @id @default(cuid())
  tenantId          String

  // Tag details
  name              String
  slug              String
  color             String?           // Hex color code
  description       String?           @db.Text

  // Categorization
  category          String?           // For grouping tags

  // Usage tracking
  usageCount        Int               @default(0)

  // Status
  isActive          Boolean           @default(true)

  // Audit
  createdById       String
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  assignments       TagAssignment[]

  @@unique([tenantId, slug])
  @@index([tenantId])
  @@map("tags")
}

model TagAssignment {
  id                String            @id @default(cuid())
  tenantId          String
  tagId             String

  // Entity association (polymorphic)
  entityType        String            // contract, invoice, expense, document, etc.
  entityId          String

  // Audit
  assignedById      String
  assignedAt        DateTime          @default(now())

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  tag               Tag               @relation(fields: [tagId], references: [id], onD
elete: Cascade)

  @@unique([tagId, entityType, entityId])
  @@index([tenantId])
  @@index([entityType, entityId])
  @@map("tag_assignments")
}
```

Task 70-71: Custom Fields and User Activity

Purpose: Allow flexible custom fields per entity and track user activities

```
model CustomField {
  id                String            @id @default(cuid())
  tenantId          String

  // Field definition
  name              String
  key               String            // machine-readable key
  fieldType         String            // text, number, date, boolean, select, multi_se
lect

  // Entity type this field applies to
  entityType        String            // contract, invoice, expense, etc.

  // Configuration
  isRequired        Boolean           @default(false)
  isActive          Boolean           @default(true)

  // Options (for select/multi_select types)
  options           Json?             // Array of options

  // Validation
  validationRules   Json?             // Custom validation rules

  // Display
  order             Int               @default(0)
  placeholder       String?
  helpText          String?

  // Audit
  createdById       String
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  values            CustomFieldValue[]

  @@unique([tenantId, entityType, key])
  @@index([tenantId])
  @@index([entityType])
  @@map("custom_fields")
}

model CustomFieldValue {
  id                String            @id @default(cuid())
  tenantId          String
  customFieldId     String

  // Entity association
  entityType        String
  entityId          String

  // Value storage
  textValue         String?           @db.Text
  numberValue       Decimal?          @db.Decimal(20, 6)
  dateValue         DateTime?
  booleanValue      Boolean?
  jsonValue         Json?             // For complex types

  // Audit
  createdAt         DateTime          @default(now())
```

```
  updatedAt         DateTime          @updatedAt

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)
  customField       CustomField       @relation(fields: [customFieldId], references:
[id], onDelete: Cascade)

  @@unique([customFieldId, entityType, entityId])
  @@index([tenantId])
  @@index([entityType, entityId])
  @@map("custom_field_values")
}

model UserActivity {
  id                String            @id @default(cuid())
  tenantId          String
  userId            String

  // Activity details
  action            String            // viewed, created, updated, deleted, exported,
etc.
  entityType        String            // contract, invoice, user, etc.
  entityId          String?
  entityName        String?           // Denormalized for quick display

  // Description
  description       String

  // Additional context
  metadata          Json?

  // Session info
  ipAddress         String?
  userAgent         String?

  // Timestamp
  occurredAt        DateTime          @default(now())

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)

  @@index([tenantId])
  @@index([userId])
  @@index([entityType, entityId])
  @@index([occurredAt])
  @@map("user_activities")
}
```

Task 73: API Key Management Table

Purpose: Manage API keys for third-party integrations and external access

```
model ApiKey {
  id                String            @id @default(cuid())
  tenantId          String

  // Key details
  name              String
  description       String?           @db.Text
  key               String            @unique // Hashed API key
  keyPrefix         String            // First 8 chars for display (e.g., "sk_live_")

  // Permissions
  scopes            String[]          // Array of allowed scopes/permissions

  // Usage limits
  rateLimit         Int?              // Requests per hour
  usageCount        Int               @default(0)
  lastUsedAt        DateTime?

  // Status
  isActive          Boolean           @default(true)
  expiresAt         DateTime?

  // IP restrictions
  allowedIPs        String[]          // Array of allowed IP addresses

  // Metadata
  metadata          Json?

  // Audit
  createdById       String
  createdAt         DateTime          @default(now())
  updatedAt         DateTime          @updatedAt
  revokedAt         DateTime?
  revokedById       String?

  // Relations
  tenant            Tenant            @relation(fields: [tenantId], references: [id],
onDelete: Cascade)

  @@index([tenantId])
  @@index([keyPrefix])
  @@index([isActive])
  @@map("api_keys")
}
```

🚀 Implementation Steps

Step 1: Database Schema Updates (Week 1, Days 1-3)

1.1 Update User Model

- Add new fields to User model in `prisma/schema.prisma`
- Add new relations
- Add new indexes

1.2 Create Core Financial Tables

- Payment table

- PaymentMethod table
- Expense table
- Update existing Invoice model to add Payment relation

1.3 Create Time Tracking Tables

- Timesheet table
- TimesheetEntry table

1.4 Create Approval System Tables

- ApprovalWorkflow table
- ApprovalStep table

1.5 Run Migrations

```
# Generate migration
npx prisma migrate dev --name phase2_database_enhancements

# Generate Prisma client
npx prisma generate

# Update seed script if needed
yarn run seed
```

Step 2: Document Management & Metadata (Week 1, Days 4-5)

2.1 Create Document Tables

- Document table (generic, beyond ContractDocument)
- Comment table
- Update relations

2.2 Create Categorization Tables

- Tag table
- TagAssignment table

2.3 Create Custom Fields Tables

- CustomField table
- CustomFieldValue table

Step 3: Activity Tracking & API Management (Week 2, Days 1-2)

3.1 Create Activity Tracking

- UserActivity table
- Add relation to User model

3.2 Create API Key Management

- ApiKey table
- Add relation to User/Tenant

Step 4: Update Existing Models (Week 2, Days 3-4)

4.1 Add Missing Relations

- Update Tenant model to include new relations
- Update Contract, Invoice, Contractor models with new relations

4.2 Add Indexes

- Add single-column indexes
- Add composite indexes for query optimization
- Add unique constraints

4.3 Update seed script

```
// scripts/seed.ts
// Add seed data for:
// - Tags
// - CustomFields
// - Payment methods (sample)
```

Step 5: API Development (Week 2-3)

For each new table, create tRPC routers with standard CRUD operations:

5.1 Payment Router (`server/api/routers/payment.ts`)

```
- getAll (with filters: status, invoiceId, dateRange)
- getById
- create
- update
- delete (soft delete)
- process (trigger payment processing)
- refund
- getByInvoice
- getByExpense
- getStatistics
```

5.2 PaymentMethod Router (`server/api/routers/paymentMethod.ts`)

```
- getAll (by owner)
- getById
- create
- update
- delete
- setDefault
- verify
```

5.3 Expense Router (`server/api/routers/expense.ts`)

```
- getAll (with filters: status, contractor, dateRange)
- getById
- create
- update
- delete
- submit (for approval)
- approve
- reject
- getByContractor
- getStatistics
```

5.4 Timesheet Router (`server/api/routers/timesheet.ts`)

```
- getAll
- getById
- create
- update
- delete
- submit
- approve
- reject
- addEntry
- updateEntry
- deleteEntry
- calculate (calculate totals)
- getByContractor
- getByContract
```

5.5 ApprovalWorkflow Router (`server/api/routers/approvalWorkflow.ts`)

```
- getAll
- getById
- create
- update
- cancel
- processStep
- approve
- reject
- getByEntity
- getPendingApprovals (for current user)
```

5.6 Document Router (`server/api/routers/document.ts`)

```
- getAll
- getById
- create (with file upload)
- update
- delete
- getByEntity
- createVersion
- sign
- getVersionHistory
```

5.7 Comment Router (`server/api/routers/comment.ts`)

```
- getAll
- getById
- create
- update
- delete
- getByEntity
- getReplies
```

5.8 Tag Router (`server/api/routers/tag.ts`)

```
- getAll
- getById
- create
- update
- delete
- assignToEntity
- removeFromEntity
- getByEntity
```

5.9 CustomField Router (`server/api/routers/customField.ts`)

```
- getAll (by entity type)
- getById
- create
- update
- delete
- getByEntityType
- setValue
- getValue
- getValuesByEntity
```

5.10 UserActivity Router (`server/api/routers/userActivity.ts`)

```
- getAll
- getById
- getByUser
- getByEntity
- getRecent
- log (internal use)
```

5.11 ApiKey Router (`server/api/routers/apiKey.ts`)

```
- getAll
- getById
- create (returns unhashed key once)
- update
- delete
- revoke
- regenerate
```

🧪 Testing Strategy

Unit Tests

For each router, create tests:

```
// Example: server/api/routers/__tests__/payment.test.ts
describe('Payment Router', () => {
  describe('getAll', () => {
    it('should return all payments for tenant')
    it('should filter by status')
    it('should filter by invoiceId')
    it('should paginate results')
  })

  describe('create', () => {
    it('should create a new payment')
    it('should validate required fields')
    it('should check permissions')
  })

  // ... more tests
})
```

Integration Tests

Test workflows:
- Complete expense submission → approval → payment flow
- Timesheet submission → approval → invoice generation
- Document upload → tagging → commenting

Database Migration Tests

```
# Test migration up
npx prisma migrate dev

# Test migration rollback if needed
npx prisma migrate reset

# Verify data integrity
```

📅 Timeline & Milestones

Week 1: Database Schema & Core Tables

Days 1-3: Core Financial & Time Tracking
- [ ] Update User model with new fields
- [ ] Create Payment, PaymentMethod, Expense tables
- [ ] Create Timesheet, TimesheetEntry tables
- [ ] Create ApprovalWorkflow, ApprovalStep tables
- [ ] Run migrations
- [ ] Update seed script

Days 4-5: Document Management
- [ ] Create Document, Comment tables
- [ ] Create Tag, TagAssignment tables
- [ ] Create CustomField, CustomFieldValue tables
- [ ] Run migrations
- [ ] Test data relationships

Week 2: API Development (Part 1)

Days 1-2: Financial APIs
- [ ] Payment router with all endpoints
- [ ] PaymentMethod router
- [ ] Expense router
- [ ] Write unit tests

Days 3-4: Time Tracking & Approvals
- [ ] Timesheet router
- [ ] ApprovalWorkflow router
- [ ] Write unit tests
- [ ] Integration tests for approval workflows

Day 5: Document Management APIs
- [ ] Document router
- [ ] Comment router
- [ ] Tag router

Week 3: API Development (Part 2) & Polish

Days 1-2: Metadata & Activity
- [ ] CustomField router
- [ ] UserActivity router
- [ ] ApiKey router
- [ ] Write unit tests

Days 3-4: Integration & Testing
- [ ] Integration tests for all workflows
- [ ] Performance testing
- [ ] Security audit of new APIs
- [ ] Update API documentation

Day 5: Final Review & Deployment
- [ ] Code review
- [ ] Documentation update
- [ ] Migration guide
- [ ] Deploy to dev branch
- [ ] Create PR for testing

✅ Success Criteria

Database Schema

- [ ] All 30 tasks from Phase 2 completed
- [ ] Zero migration errors
- [ ] All foreign keys properly configured
- [ ] All indexes created and tested
- [ ] Seed script works without errors

API Development

- [ ] All 11 routers implemented
- [ ] All endpoints have proper permission checks
- [ ] All endpoints have input validation
- [ ] All endpoints have error handling
- [ ] Unit test coverage > 80%

Performance

- [ ] Query performance tested with 10k+ records
- [ ] All indexed queries under 100ms
- [ ] Complex queries under 500ms
- [ ] No N+1 query issues

Documentation

- [ ] All new tables documented
- [ ] All API endpoints documented
- [ ] Migration guide created
- [ ] Seed data examples provided

Security

- [ ] All endpoints require authentication
- [ ] All endpoints check tenant isolation
- [ ] All endpoints validate permissions
- [ ] Sensitive data properly encrypted
- [ ] SQL injection prevention verified

📚 Additional Notes

Data Migration Considerations

If migrating from existing system:
1. Create migration scripts for existing data
2. Map old schema to new schema
3. Validate data integrity after migration
4. Create rollback plan

Performance Optimization Tips

1. **Use select judiciously**: Only fetch needed fields

```
select: {
  id: true,
  name: true,
  // Don't fetch entire related objects unless needed
}
```

1. **Implement pagination everywhere**

```
take: input?.limit ?? 50,
skip: input?.offset ?? 0,
```

1. **Use database-level calculations** when possible

```
// Use Prisma aggregations instead of fetching all and calculating in JS
const { _sum } = await prisma.payment.aggregate({
  _sum: { amount: true },
  where: { tenantId, status: 'completed' }
})
```

1. **Cache frequently accessed data**

```
// Use Redis for caching
const cached = await redis.get(`tenant:${tenantId}:stats`)
```

Security Best Practices

1. **Always validate tenant isolation**

```
where: {
  id: input.id,
  tenantId: ctx.tenantId // CRITICAL: Always include tenantId
}
```

1. **Encrypt sensitive data**

```
// For payment methods, bank accounts, etc.
const encrypted = await encrypt(sensitiveData)
```

1. **Implement rate limiting**

```
// Use rate-limit middleware for sensitive endpoints
.use(rateLimitMiddleware({ maxRequests: 100, windowMs: 60000 }))
```

1. **Audit trail for sensitive operations**

```
await createAuditLog({
  action: 'payment_processed',
  entityType: 'payment',
  entityId: payment.id,
  // ...
})
```

🔄 Next Steps After Phase 2

Once Phase 2 is complete, the foundation will be ready for:

- **Phase 3:** Multi-tenancy & White-label features

- **Phase 6:** UI/UX improvements and missing pages
- **Phase 7:** Contract & Payroll workflows (depends heavily on Phase 2)
- **Phase 8:** Notification systems

📞 Support & Questions

For questions during implementation:
- Review existing router patterns in `server/api/routers/`
- Check Prisma documentation for complex queries
- Refer to existing test examples
- Review RBAC implementation in `server/rbac/`

Document Version: 1.0
Last Updated: November 15, 2025
Status: Ready for Implementation
Estimated Completion: 2-3 weeks